



Intel® Math Kernel Library

Reference Manual

Document Number: 630813-065US

MKL 11.2

[Legal Information](#)

Contents

Legal Information.....	31
Introducing the Intel® Math Kernel Library.....	35
Getting Help and Support.....	37
What's New.....	39
Notational Conventions.....	41
Chapter 1: Function Domains	
Performance Enhancements.....	47
Parallelism.....	48
C Datatypes Specific to Intel MKL.....	49
Chapter 2: BLAS and Sparse BLAS Routines	
BLAS Routines.....	51
Routine Naming Conventions.....	51
Fortran 95 Interface Conventions.....	53
C Interface Conventions.....	53
Matrix Storage Schemes.....	54
BLAS Level 1 Routines and Functions.....	55
?asum.....	56
?axpy.....	57
?copy.....	59
?dot.....	60
?sdot.....	61
?dotc.....	63
?dotu.....	64
?nrm2.....	65
?rot.....	67
?rotg.....	68
?rotm.....	70
?rotmg.....	72
?scal.....	74
?swap.....	75
i?amax.....	77
i?amin.....	78
?cabs1.....	79
BLAS Level 2 Routines.....	80
?gbmv.....	81
?gemv.....	84
?ger.....	87
?gerc.....	89
?geru.....	91
?hbmv.....	93
?hemv.....	96
?her.....	98
?her2.....	100

?hpmv.....	102
?hpr.....	104
?hpr2.....	106
?sbmv.....	108
?spmv.....	111
?spr.....	113
?spr2.....	115
?symv.....	117
?syr.....	119
?syr2.....	121
?tbmv.....	123
?tbsv.....	126
?tpmv.....	129
?tpsv.....	131
?trmv.....	134
?trsv.....	136
BLAS Level 3 Routines.....	138
?gemm.....	139
?hemm.....	143
?herk.....	146
?her2k.....	148
?symm.....	151
?syrk.....	155
?syr2k.....	158
?trmm.....	161
?trsm.....	164
Sparse BLAS Level 1 Routines.....	167
Vector Arguments.....	167
Naming Conventions.....	168
Routines and Data Types.....	168
BLAS Level 1 Routines That Can Work With Sparse Vectors.....	168
?axpyi.....	169
?doti.....	171
?dotci.....	172
?dotui.....	173
?gthr.....	175
?gthrz.....	176
?roti.....	177
?sctr.....	179
Sparse BLAS Level 2 and Level 3 Routines.....	180
Naming Conventions in Sparse BLAS Level 2 and Level 3.....	181
Sparse Matrix Storage Formats.....	182
Routines and Supported Operations.....	182
Interface Consideration.....	183
Sparse BLAS Level 2 and Level 3 Routines.....	188
mkl_?csrgemv.....	191
mkl_?bsrgemv.....	194
mkl_?coogemv.....	197
mkl_?diagemv.....	200
mkl_?csrsvmv.....	203

mkl_?bsrsymv.....	206
mkl_?coosymv.....	209
mkl_?diasymv.....	212
mkl_?csrtrsv.....	215
mkl_?bsrtrsv.....	218
mkl_?cootrv.....	221
mkl_?diatrv.....	224
mkl_cspblas_?csrgemv.....	228
mkl_cspblas_?bsrgemv.....	231
mkl_cspblas_?coogemv.....	234
mkl_cspblas_?csrsv.....	237
mkl_cspblas_?bsrsymv.....	240
mkl_cspblas_?coosymv.....	243
mkl_cspblas_?csrtrsv.....	246
mkl_cspblas_?bsrtrsv.....	249
mkl_cspblas_?cootrv.....	252
mkl_?csrmv.....	255
mkl_?bsrmv.....	259
mkl_?cscmv.....	264
mkl_?coomv.....	268
mkl_?csrv.....	272
mkl_?bsrsv.....	276
mkl_?cscsv.....	280
mkl_?coosv.....	284
mkl_?csrmm.....	288
mkl_?bsrmm.....	292
mkl_?cscmm.....	297
mkl_?coomm.....	302
mkl_?csrsm.....	306
mkl_?cscsm.....	310
mkl_?coosm.....	315
mkl_?bsrsm.....	318
mkl_?diamv.....	322
mkl_?skymv.....	326
mkl_?diasv.....	330
mkl_?skysv.....	334
mkl_?diamm.....	337
mkl_?skymm.....	341
mkl_?diasm.....	346
mkl_?skysm.....	350
mkl_?dnscsr.....	353
mkl_?csrcoo.....	357
mkl_?csrbsr.....	360
mkl_?csrcsc.....	364
mkl_?csrdia.....	367
mkl_?csrsky.....	371
mkl_?csradd.....	375
mkl_?csrmultcsr.....	379
mkl_?csrmultd.....	384
BLAS-like Extensions.....	387

?axpby.....	388
?gem2vu.....	390
?gem2vc.....	392
?gemm3m.....	395
mkl_?imatcopy.....	398
mkl_?omatcopy.....	401
mkl_?omatcopy2.....	404
mkl_?omatadd.....	407

Chapter 3: LAPACK Routines

Routine Naming Conventions.....	411
C Interface Conventions.....	412
Fortran 95 Interface Conventions.....	415
Intel® MKL Fortran 95 Interfaces for LAPACK Routines vs. Netlib Implementation.....	416
Matrix Storage Schemes.....	417
Mathematical Notation.....	417
Error Analysis.....	418
Linear Equation Routines.....	419
Computational Routines.....	419
Routines for Matrix Factorization.....	421
Routines for Solving Systems of Linear Equations.....	459
Routines for Estimating the Condition Number.....	502
Refining the Solution and Estimating Its Error.....	537
Routines for Matrix Inversion.....	611
Routines for Matrix Equilibration.....	642
Driver Routines.....	664
?gesv.....	665
?gesvx.....	669
?gesvxx.....	675
?gbsv.....	684
?gbsvx.....	686
?gbsvxx.....	692
?gtsv.....	701
?gtsvx.....	702
?dttsb.....	707
?posv.....	709
?posvx.....	713
?posvxx.....	718
?ppsv.....	726
?ppsvx.....	728
?pbsv.....	733
?pbsvx.....	735
?ptsv.....	740
?ptsvx.....	742
?sysv.....	746
?sysv_rook.....	749
?sysvx.....	751
?sysvxx.....	756
?hesv.....	764

?hesv_rook.....	767
?hesvx.....	769
?hesvxx.....	774
?spsv.....	781
?spsvx.....	783
?hpsv.....	788
?hpsvx.....	790
Least Square and Eigenvalue Problem Routines.....	793
Computational Routines.....	794
Orthogonal Factorizations.....	795
Singular Value Decomposition - LAPACK Computational Routines.	878
Symmetric Eigenvalue Problems.....	905
Generalized Symmetric-Definite Eigenvalue Problems.....	975
Nonsymmetric Eigenvalue Problems.....	990
Generalized Nonsymmetric Eigenvalue Problems.....	1041
Generalized Singular Value Decomposition.....	1079
Cosine-Sine Decomposition.....	1091
Driver Routines.....	1103
Linear Least Squares (LLS) Problems.....	1104
Generalized LLS Problems.....	1118
Symmetric Eigenproblems.....	1124
Nonsymmetric Eigenproblems.....	1208
Singular Value Decomposition - LAPACK Driver Routines.....	1229
Cosine-Sine Decomposition.....	1255
Generalized Symmetric Definite Eigenproblems.....	1265
Generalized Nonsymmetric Eigenproblems.....	1331
Auxiliary Routines.....	1358
?lacgv.....	1371
?lacrm.....	1371
?lacrt.....	1372
?laesy.....	1373
?rot.....	1375
?spmv.....	1376
?spr.....	1377
?syconv.....	1379
?symv.....	1380
?syr.....	1382
i?max1.....	1383
?sum1.....	1384
?gbtf2.....	1385
?gebd2.....	1386
?gehd2.....	1388
?gelq2.....	1390
?geql2.....	1391
?geqr2.....	1393
?geqr2p.....	1394
?geqrt2.....	1396
?geqrt3.....	1398
?gerq2.....	1400
?gesc2.....	1401

?getc2.....	1402
?getf2.....	1404
?gtts2.....	1405
?isnan.....	1406
?laisnan.....	1407
?labrd.....	1407
?lacn2.....	1410
?lacon.....	1412
?lacpy.....	1413
?ladiv.....	1414
?lae2.....	1415
?laebz.....	1416
?laed0.....	1420
?laed1.....	1423
?laed2.....	1424
?laed3.....	1426
?laed4.....	1428
?laed5.....	1430
?laed6.....	1431
?laed7.....	1432
?laed8.....	1435
?laed9.....	1438
?laeda.....	1440
?laein.....	1442
?laev2.....	1444
?laexc.....	1446
?lag2.....	1447
?lags2.....	1449
?lagtf.....	1452
?lagtm.....	1454
?lagts.....	1455
?lagv2.....	1457
?lahqr.....	1459
?lahrd.....	1461
?lahr2.....	1464
?laic1.....	1466
?laln2.....	1468
?lals0.....	1470
?lalsa.....	1474
?lalsd.....	1477
?lamrg.....	1479
?laneg.....	1480
?langb.....	1481
?lange.....	1483
?langt.....	1484
?lanhs.....	1485
?lansb.....	1486
?lanhb.....	1488
?lansp.....	1489
?lanhp.....	1491

?lanst/?lanht.....	1492
?lansy.....	1493
?lanhe.....	1495
?lantb.....	1496
?lantp.....	1498
?lantr.....	1500
?lanv2.....	1502
?lapll.....	1503
?lapmr.....	1504
?lapmt.....	1505
?lapy2.....	1506
?lapy3.....	1507
?laqgb.....	1508
?laqge.....	1509
?laqhb.....	1511
?laqp2.....	1513
?laqps.....	1514
?laqr0.....	1516
?laqr1.....	1519
?laqr2.....	1520
?laqr3.....	1524
?laqr4.....	1527
?laqr5.....	1531
?laqsb.....	1534
?laqsp.....	1535
?laqsy.....	1537
?laqtr.....	1538
?lar1v.....	1541
?lar2v.....	1544
?larf.....	1545
?larfb.....	1547
?larfg.....	1550
?larfgp.....	1551
?larft.....	1553
?larfx.....	1555
?largv.....	1557
?larnv.....	1558
?larra.....	1560
?larrb.....	1561
?larrc.....	1563
?larrd.....	1564
?larre.....	1568
?larrf.....	1571
?larrj.....	1573
?larrk.....	1575
?larrr.....	1577
?larry.....	1578
?lartg.....	1581
?lartgp.....	1583
?lartgs.....	1584

?lartv.....	1586
?laruv.....	1587
?larz.....	1588
?larzb.....	1590
?larzt.....	1592
?las2.....	1595
?lascl.....	1595
?lasd0.....	1597
?lasd1.....	1598
?lasd2.....	1601
?lasd3.....	1604
?lasd4.....	1607
?lasd5.....	1608
?lasd6.....	1609
?lasd7.....	1613
?lasd8.....	1617
?lasd9.....	1619
?lasda.....	1621
?lasdq.....	1624
?lasdt.....	1627
?laset.....	1627
?lasq1.....	1629
?lasq2.....	1630
?lasq3.....	1631
?lasq4.....	1633
?lasq5.....	1634
?lasq6.....	1636
?lasr.....	1637
?lasrt.....	1640
?lassq.....	1641
?lasv2.....	1642
?laswp.....	1643
?lasy2.....	1644
?lasyf.....	1647
?lasyf_rook.....	1649
?lahef.....	1651
?lahef_rook.....	1653
?latbs.....	1655
?latdf.....	1657
?latps.....	1659
?latrd.....	1661
?latrs.....	1663
?latrz.....	1667
?lauu2.....	1669
?lauum.....	1670
?orbdb1/?unbdb1.....	1671
?orbdb2/?unbdb2.....	1674
?orbdb3/?unbdb3.....	1678
?orbdb4/?unbdb4.....	1681
?orbdb5/?unbdb5.....	1684

?orbdb6/?unbdb6.....	1687
?org2l/?ung2l.....	1689
?org2r/?ung2r.....	1691
?orgl2/?ungl2.....	1692
?orgr2/?ungr2.....	1694
?orm2l/?unm2l.....	1695
?orm2r/?unm2r.....	1697
?orml2/?unml2.....	1699
?ormr2/?unmr2.....	1701
?ormr3/?unmr3.....	1703
?pbtf2.....	1706
?potf2.....	1707
?ptts2.....	1708
?rscl.....	1710
?syswapr.....	1711
?heswapr.....	1712
?sygs2/?hegs2.....	1715
?sytd2/?hetd2.....	1717
?sytf2.....	1719
?sytf2_rook.....	1721
?hetf2.....	1722
?hetf2_rook.....	1724
?tgex2.....	1725
?tgpsy2.....	1727
?trti2.....	1731
clag2z.....	1732
dlag2s.....	1733
slag2d.....	1734
zlag2c.....	1735
?larfp.....	1736
ila?lc.....	1737
ila?lr.....	1738
?gsvj0.....	1739
?gsvj1.....	1742
?sfrk.....	1744
?hfrk.....	1746
?tfsm.....	1748
?lansf.....	1750
?lanhf.....	1752
?tfttp.....	1753
?tftrr.....	1754
?tpqrt2	1756
?tprb	1759
?tpttf.....	1762
?tpttr.....	1764
?trttf.....	1765
?trttp.....	1767
?pstf2.....	1768
dlat2s	1770
zlat2c	1771

?lacp2.....	1772
?la_gbamv.....	1773
?la_gbrcond.....	1775
?la_gbrcond_c.....	1777
?la_gbrcond_x.....	1779
?la_gbrfsx_extended.....	1781
?la_gbrpvgrw.....	1787
?la_geamv.....	1788
?la_gercond.....	1790
?la_gercond_c.....	1792
?la_gercond_x.....	1793
?la_gerfsx_extended.....	1794
?la_heamv.....	1800
?la_hercond_c.....	1802
?la_hercond_x.....	1804
?la_herfsx_extended.....	1805
?la_herpvgrw.....	1811
?la_lin_berr.....	1812
?la_porcond.....	1813
?la_porcond_c.....	1815
?la_porcond_x.....	1816
?la_porfsx_extended.....	1818
?la_porpvgrw.....	1824
?laqhe.....	1825
?laqhp.....	1827
?larcm.....	1828
?la_gerpvgrw.....	1829
?larscl2.....	1830
?lascl2.....	1831
?la_syamv.....	1832
?la_syrcond.....	1834
?la_syrcond_c.....	1836
?la_syrcond_x.....	1837
?la_syrsx_extended.....	1839
?la_syrpvgrw.....	1845
?la_wwaddw.....	1847
Utility Functions and Routines.....	1847
ilaver.....	1848
ilaenv.....	1849
iparmq.....	1851
ieecek.....	1853
?labad.....	1854
?lamch.....	1855
?lamc1.....	1856
?lamc2.....	1856
?lamc3.....	1857
?lamc4.....	1858
?lamc5.....	1859
chl_a_transtype.....	1859
iladiag.....	1860

ilaprec.....	1861
ilatrans.....	1861
ilauplo.....	1862
xerbla_array.....	1863
Test Functions and Routines.....	1864
?lagge.....	1864
?laghe.....	1865
?lagsy.....	1866
?latms.....	1867

Chapter 4: ScaLAPACK Routines

Overview.....	1873
Routine Naming Conventions.....	1875
Computational Routines.....	1876
Linear Equations.....	1876
Routines for Matrix Factorization.....	1877
p?getrf.....	1877
p?gbtrf.....	1879
p?dbtrf.....	1882
p?dttrf.....	1884
p?potrf.....	1886
p?pbtrf.....	1888
p?pttrf.....	1890
Routines for Solving Systems of Linear Equations.....	1893
p?getrs.....	1893
p?gbtrs.....	1895
p?dbtrs.....	1898
p?dttrs.....	1900
p?potrs.....	1903
p?pbtrs.....	1905
p?pttrs.....	1907
p?trtrs.....	1909
Routines for Estimating the Condition Number.....	1912
p?gecon.....	1912
p?pocon.....	1915
p?trcon.....	1918
Refining the Solution and Estimating Its Error.....	1921
p?gerfs.....	1921
p?porfs.....	1925
p?ttrfs.....	1928
Routines for Matrix Inversion.....	1932
p?getri.....	1932
p?potri.....	1935
p?trtri.....	1936
Routines for Matrix Equilibration.....	1938
p?geequ.....	1938
p?poequ.....	1940
Orthogonal Factorizations.....	1943
p?geqrf.....	1943
p?geqpf.....	1946

p?orgqr.....	1948
p?ungqr.....	1950
p?ormqr.....	1952
p?unmqr.....	1955
p?gelqf.....	1958
p?orglq.....	1961
p?unglq.....	1963
p?ormlq.....	1965
p?unmlq.....	1968
p?geqlf.....	1971
p?orgql.....	1973
p?ungql.....	1975
p?ormql.....	1977
p?unmql.....	1980
p?gerqf.....	1983
p?orgrq.....	1986
p?ungrq.....	1988
p?ormrq.....	1989
p?unmrq.....	1992
p?tzrzf.....	1996
p?ormrz.....	1998
p?unmrz.....	2002
p?ggqrf.....	2005
p?ggrqf.....	2009
Symmetric Eigenproblems.....	2013
p?syrtd.....	2014
p?ormtr.....	2017
p?hetrd.....	2021
p?unmtr.....	2024
p?stebz.....	2027
p?stein.....	2031
Nonsymmetric Eigenvalue Problems.....	2035
p?gehrd.....	2035
p?ormhr.....	2038
p?unmhr.....	2042
p?lahqr.....	2045
p?hseqr.....	2047
Singular Value Decomposition.....	2050
p?gebrd.....	2050
p?ormbr.....	2054
p?unmbr.....	2058
Generalized Symmetric-Definite Eigen Problems.....	2063
p?sygst.....	2063
p?hegst.....	2065
Driver Routines.....	2067
p?gesv.....	2067
p?gesvx.....	2069
p?gbsv.....	2075
p?dbsv.....	2078
p?dtsv.....	2080

p?posv.....	2083
p?posvx.....	2085
p?pbsv.....	2091
p?ptsv.....	2094
p?gels.....	2097
p?syev.....	2100
p?syevd.....	2103
p?syevr.....	2106
p?syevx.....	2110
p?heev.....	2116
p?heevd.....	2120
p?heevr.....	2123
p?heevx.....	2128
p?gesvd.....	2135
p?sygvx.....	2139
p?hegvx.....	2146

Chapter 5: ScaLAPACK Auxiliary, Utility, and Redistribution/Copy Routines

Auxiliary Routines.....	2155
b?laapp.....	2160
b?laexc.....	2162
b?trexc.....	2163
p?lacgv.....	2166
p?max1.....	2167
pmpcol.....	2168
pmpim2.....	2169
?combamax1.....	2170
p?sum1.....	2171
p?dbtrsv.....	2172
p?dttrsv.....	2175
p?gebal.....	2179
p?gebd2.....	2181
p?gehd2.....	2184
p?gelq2.....	2187
p?geql2.....	2190
p?geqr2.....	2192
p?gerq2.....	2194
p?getf2.....	2197
p?labrd.....	2199
p?lacon.....	2203
p?laconsb.....	2205
p?lacp2.....	2206
p?lacp3.....	2208
p?lacpy.....	2209
p?laevswp.....	2211
p?lahrd.....	2213
p?laiect.....	2216
p?lamve.....	2217
p?lange.....	2219

p?lanhs.....	2221
p?lansy, p?lanhe.....	2223
p?lantr.....	2225
p?lapiv.....	2227
p?laqge.....	2230
p?laqr0.....	2233
p?laqr1.....	2235
p?laqr2.....	2238
p?laqr3.....	2241
p?laqr4.....	2244
p?laqr5.....	2247
p?laqsy.....	2249
p?lared1d.....	2251
p?lared2d.....	2253
p?larf.....	2254
p?larfb.....	2257
p?larfc.....	2261
p?larfg.....	2264
p?larft.....	2266
p?larz.....	2269
p?larzb.....	2273
p?larzc.....	2277
p?larzt.....	2280
p?lascl.....	2283
p?laset.....	2285
p?lasmsub.....	2287
p?lassq.....	2288
p?laswp.....	2290
p?latra.....	2292
p?latrd.....	2293
p?latrs.....	2297
p?latrz.....	2300
p?lauu2.....	2302
p?lauum.....	2304
p?lawil.....	2305
p?org2l/p?ung2l.....	2307
p?org2r/p?ung2r.....	2309
p?orgl2/p?unql2.....	2312
p?orgr2/p?ungr2.....	2314
p?orm2l/p?unm2l.....	2317
p?orm2r/p?unm2r.....	2320
p?orml2/p?unml2.....	2324
p?ormr2/p?unmr2.....	2328
p?pbtrsv.....	2332
p?pttrsv.....	2336
p?potf2.....	2340
p?rot.....	2341
p?rscl.....	2344
p?sygs2/p?hegs2.....	2345
p?syt2/p?hetd2.....	2348

p?trord.....	2351
p?trsen.....	2355
p?trti2.....	2359
?lamsh.....	2361
?laqr6.....	2363
?lar1va.....	2366
?laref.....	2369
?larrb2.....	2371
?larrd2.....	2374
?larre2.....	2377
?larre2a.....	2381
?larrf2.....	2386
?larry2.....	2388
?lasorte.....	2393
?lasrt2.....	2394
?stegr2.....	2395
?stegr2a.....	2399
?stegr2b.....	2402
?stein2.....	2406
?dbtf2.....	2408
?dbtrf.....	2410
?dttrf.....	2412
?dttrsv.....	2413
?pttrsv.....	2415
?steqr2.....	2416
Utility Functions and Routines.....	2418
p?labad.....	2419
p?lachkieee.....	2420
p?lamch.....	2421
p?lasnbt.....	2422
Matrix Redistribution/Copy Routines.....	2423
p?gemr2d.....	2423
p?trmr2d.....	2425

Chapter 6: Sparse Solver Routines

Intel MKL PARDISO - Parallel Direct Sparse Solver Interface.....	2429
pardiso.....	2434
pardisoinit.....	2440
pardiso_64.....	2442
pardiso_getenv, pardiso_setenv.....	2443
mkl_pardiso_pivot.....	2444
pardiso_getdiag.....	2445
pardiso_handle_store.....	2447
pardiso_handle_restore.....	2448
pardiso_handle_delete.....	2449
pardiso_handle_store_64.....	2450
pardiso_handle_restore_64.....	2451
pardiso_handle_delete_64.....	2452
Intel MKL PARDISO Parameters in Tabular Form.....	2452
pardiso iparm Parameter.....	2457

PARDISO_DATA_TYPE.....	2467
Parallel Direct Sparse Solver for Clusters Interface.....	2467
cluster_sparse_solver.....	2469
cluster_sparse_solver iparm Parameter.....	2474
Direct Sparse Solver (DSS) Interface Routines.....	2478
DSS Interface Description.....	2480
DSS Implementation Details.....	2481
DSS Routines.....	2482
dss_create.....	2482
dss_define_structure.....	2483
dss_reorder.....	2485
dss_factor_real, dss_factor_complex.....	2487
dss_solve_real, dss_solve_complex.....	2489
dss_delete.....	2492
dss_statistics.....	2493
mkl_cvt_to_null_terminated_str.....	2496
Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS).....	2496
CG Interface Description.....	2498
FGMRES Interface Description.....	2503
RCI ISS Routines.....	2511
dcg_init.....	2511
dcg_check.....	2512
dcg.....	2513
dcg_get.....	2515
dcgmrhs_init.....	2516
dcgmrhs_check.....	2517
dcgmrhs.....	2518
dcgmrhs_get.....	2520
dfgmres_init.....	2521
dfgmres_check.....	2522
dfgmres.....	2523
dfgmres_get.....	2525
Implementation Details.....	2526
Preconditioners based on Incomplete LU Factorization Technique.....	2527
ILU0 and ILUT Preconditioners Interface Description.....	2529
dcsrilu0.....	2530
dcsrilut.....	2533
Sparse Matrix Checker Routines.....	2537
sparse_matrix_checker.....	2537
sparse_matrix_init.....	2539
Calling Sparse Solver and Preconditioner Routines from C and C+ +.....	2541

Chapter 7: Extended Eigensolver Routines

The FEAST Algorithm.....	2543
Extended Eigensolver Functionality.....	2544
Parallelism in Extended Eigensolver Routines.....	2545
Achieving Performance With Extended Eigensolver Routines.....	2545
Extended Eigensolver Interfaces.....	2546
Extended Eigensolver Naming Conventions.....	2546

feastinit.....	2547
Extended Eigensolver Input Parameters.....	2548
Extended Eigensolver Output Details.....	2549
Extended Eigensolver RCI Routines.....	2550
Extended Eigensolver RCI Interface Description.....	2550
?feast_srcI/?feast_hrcI.....	2552
Extended Eigensolver Predefined Interfaces.....	2556
Matrix Storage.....	2556
?feast_syev/?feast_heev.....	2556
?feast_sygv/?feast_hegv.....	2559
?feast_sbev/?feast_hbev.....	2562
?feast_sbgv/?feast_hbgv.....	2565
?feast_scsvrev/?feast_hcsrev.....	2568
?feast_scsvrgv/?feast_hcsrv.....	2571

Chapter 8: Vector Mathematical Functions

Data Types, Accuracy Modes, and Performance Tips.....	2575
Function Naming Conventions.....	2576
Function Interfaces.....	2577
VML Mathematical Functions.....	2577
Pack Functions.....	2578
Unpack Functions.....	2578
Service Functions.....	2578
Input Parameters.....	2579
Output Parameters.....	2579
Vector Indexing Methods.....	2579
Error Diagnostics.....	2580
VML Mathematical Functions.....	2581
Special Value Notations.....	2582
Arithmetic Functions.....	2583
v?Add.....	2583
v?Sub.....	2585
v?Sqr.....	2588
v?Mul.....	2589
v?MulByConj.....	2592
v?Conj.....	2594
v?Abs.....	2595
v?Arg.....	2597
v?LinearFrac.....	2599
Power and Root Functions.....	2602
v?Inv.....	2602
v?Div.....	2603
v?Sqrt.....	2606
v?InvSqrt.....	2609
v?Cbrt.....	2610
v?InvCbrt.....	2612
v?Pow2o3.....	2613
v?Pow3o2.....	2615
v?Pow.....	2617
v?Powx.....	2621

v?Hypot.....	2623
Exponential and Logarithmic Functions.....	2625
v?Exp.....	2625
v?Expm1.....	2628
v?Ln.....	2630
v?Log10.....	2633
v?Log1p.....	2636
Trigonometric Functions.....	2638
v?Cos.....	2638
v?Sin.....	2640
v?SinCos.....	2642
v?CIS.....	2644
v?Tan.....	2646
v?Acos.....	2648
v?Asin.....	2651
v?Atan.....	2654
v?Atan2.....	2656
Hyperbolic Functions.....	2658
v?Cosh.....	2658
v?Sinh.....	2662
v?Tanh.....	2665
v?Acosh.....	2667
v?Asinh.....	2670
v?Atanh.....	2673
Special Functions.....	2676
v?Erf.....	2676
v?Erfc.....	2679
v?CdfNorm.....	2681
v?ErfInv.....	2683
v?ErfcInv.....	2686
v?CdfNormInv.....	2688
v?LGamma.....	2690
v?TGamma.....	2692
Rounding Functions.....	2694
v?Floor.....	2694
v?Ceil.....	2696
v?Trunc.....	2697
v?Round.....	2699
v?NearbyInt.....	2700
v?Rint.....	2702
v?Modf.....	2704
v?Frac.....	2706
VML Pack/Unpack Functions.....	2707
v?Pack.....	2708
v?Unpack.....	2711
VML Service Functions.....	2714
vmlSetMode.....	2714
vmlGetMode.....	2716
MKLFreeTIs.....	2717
vmlSetErrStatus.....	2717

vmlGetErrStatus.....	2718
vmlClearErrStatus.....	2719
vmlSetErrorCallBack.....	2719
vmlGetErrorCallBack.....	2722
vmlClearErrorCallBack.....	2722
Chapter 9: Statistical Functions	
Random Number Generators.....	2725
Conventions.....	2726
Mathematical Notation.....	2727
Naming Conventions.....	2728
Basic Generators.....	2732
BRNG Parameter Definition.....	2733
Random Streams.....	2734
Data Types.....	2735
Error Reporting.....	2735
VSL RNG Usage Model.....	2736
Service Routines.....	2738
vslNewStream.....	2739
vslNewStreamEx.....	2741
vslNewAbstractStream.....	2742
vslDNewAbstractStream.....	2744
vslsNewAbstractStream.....	2746
vslDeleteStream.....	2749
vslCopyStream.....	2749
vslCopyStreamState.....	2750
vslSaveStreamF.....	2751
vslLoadStreamF.....	2753
vslSaveStreamM.....	2754
vslLoadStreamM.....	2755
vslGetStreamSize.....	2756
vslLeapfrogStream.....	2757
vslSkipAheadStream.....	2760
vslGetStreamStateBrng.....	2762
vslGetNumRegBrngs.....	2763
Distribution Generators.....	2764
Continuous Distributions.....	2767
Discrete Distributions.....	2801
Advanced Service Routines.....	2821
Data types.....	2821
vslRegisterBrng.....	2822
vslGetBrngProperties.....	2823
Formats for User-Designed Generators.....	2824
Convolution and Correlation.....	2827
Naming Conventions.....	2828
Data Types.....	2828
Parameters.....	2829
Task Status and Error Reporting.....	2830
Task Constructors.....	2832
vslConvNewTask/vslCorrNewTask.....	2833

vslConvNewTask1D/vslCorrNewTask1D.....	2835
vslConvNewTaskX/vslCorrNewTaskX.....	2837
vslConvNewTaskX1D/vslCorrNewTaskX1D.....	2841
Task Editors.....	2844
vslConvSetMode/vslCorrSetMode.....	2845
vslConvSetInternalPrecision/vslCorrSetInternalPrecision.....	2847
vslConvSetStart/vslCorrSetStart.....	2848
vslConvSetDecimation/vslCorrSetDecimation.....	2849
Task Execution Routines.....	2851
vslConvExec/vslCorrExec.....	2851
vslConvExec1D/vslCorrExec1D.....	2855
vslConvExecX/vslCorrExecX.....	2858
vslConvExecX1D/vslCorrExecX1D.....	2862
Task Destructors.....	2866
vslConvDeleteTask/vslCorrDeleteTask.....	2866
Task Copy.....	2867
vslConvCopyTask/vslCorrCopyTask.....	2867
Usage Examples.....	2869
Mathematical Notation and Definitions.....	2871
Data Allocation.....	2872
Summary Statistics.....	2874
Naming Conventions.....	2876
Data Types.....	2876
Parameters.....	2877
Task Status and Error Reporting.....	2877
Task Constructors.....	2881
vslSSNewTask.....	2881
Task Editors.....	2883
vslISSEditTask.....	2884
vslISSEditMoments.....	2893
vslISSEditSums.....	2895
vslISSEditCovCor.....	2897
vslISSEditCP.....	2899
vslISSEditPartialCovCor.....	2901
vslISSEditQuantiles.....	2903
vslISSEditStreamQuantiles.....	2904
vslISSEditPooledCovariance.....	2906
vslISSEditRobustCovariance.....	2908
vslISSEditOutliersDetection.....	2910
vslISSEditMissingValues.....	2912
vslISSEditCorParameterization.....	2916
Task Computation Routines.....	2918
vslSSCompute.....	2921
Task Destructor.....	2922
vslSSDeleteTask.....	2922
Usage Examples.....	2923
Mathematical Notation and Definitions.....	2924
Chapter 10: Fourier Transform Functions	
FFT Functions.....	2932

FFT Interface.....	2933
Computing an FFT.....	2933
Configuration Settings.....	2934
DFTI_PRECISION.....	2936
DFTI_FORWARD_DOMAIN.....	2936
DFTI_DIMENSION, DFTI_LENGTHS.....	2938
DFTI_PLACEMENT.....	2938
DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE.....	2938
DFTI_NUMBER_OF_USER_THREADS.....	2939
DFTI_THREAD_LIMIT.....	2939
DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES.....	2940
DFTI_NUMBER_OF_TRANSFORMS.....	2942
DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE.....	2942
DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE.....	2943
DFTI_PACKED_FORMAT.....	2946
DFTI_WORKSPACE.....	2959
DFTI_COMMIT_STATUS.....	2959
DFTI_ORDERING.....	2960
Descriptor Manipulation Functions.....	2960
DftiCreateDescriptor.....	2960
DftiCommitDescriptor.....	2963
DftiFreeDescriptor.....	2964
DftiCopyDescriptor.....	2966
Descriptor Configuration Functions.....	2967
DftiSetValue.....	2967
DftiGetValue.....	2969
FFT Computation Functions.....	2971
DftiComputeForward.....	2971
DftiComputeBackward.....	2974
Configuring and Computing an FFT in Fortran.....	2977
Configuring and Computing an FFT in C/C++.....	2980
Status Checking Functions.....	2982
DftiErrorClass.....	2983
DftiErrorMessage.....	2984
Cluster FFT Functions.....	2985
Computing Cluster FFT.....	2986
Distributing Data among Processes.....	2987
Cluster FFT Interface.....	2989
Descriptor Manipulation Functions.....	2989
DftiCreateDescriptorDM.....	2990
DftiCommitDescriptorDM.....	2991
DftiFreeDescriptorDM.....	2992
FFT Computation Functions.....	2993
DftiComputeForwardDM.....	2993
DftiComputeBackwardDM.....	2995
Descriptor Configuration Functions.....	2997
DftiSetValueDM.....	2997
DftiGetValueDM.....	2999
Error Codes.....	3001

Chapter 11: PBLAS Routines

Overview.....	3003
Routine Naming Conventions.....	3004
PBLAS Level 1 Routines.....	3006
p?amax.....	3006
p?asum.....	3007
p?axpy.....	3009
p?copy.....	3010
p?dot.....	3012
p?dotc.....	3014
p?dotu.....	3015
p?nrm2.....	3017
p?scal.....	3018
p?swap.....	3019
PBLAS Level 2 Routines.....	3021
p?gemv.....	3022
p?agemv.....	3025
p?ger.....	3028
p?gerc.....	3030
p?geru.....	3032
p?hemv.....	3034
p?ahemv.....	3036
p?her.....	3038
p?her2.....	3040
p?symv.....	3042
p?asymv.....	3044
p?syr.....	3047
p?syr2.....	3048
p?trmv.....	3051
p?atrmv.....	3053
p?trsv.....	3056
PBLAS Level 3 Routines.....	3058
p?geadd.....	3059
p?tradd.....	3061
p?gemm.....	3063
p?hemm.....	3066
p?herk.....	3069
p?her2k.....	3071
p?symm.....	3073
p?syrk.....	3076
p?syr2k.....	3079
p?tran.....	3082
p?tranu.....	3083
p?tranc.....	3085
p?trmm.....	3086
p?trsm.....	3089

Chapter 12: Partial Differential Equations Support

Trigonometric Transform Routines.....	3093
---------------------------------------	------

Transforms Implemented.....	3094
Sequence of Invoking TT Routines.....	3095
Interface Description.....	3097
TT Routines.....	3097
?_init_trig_transform.....	3097
?_commit_trig_transform.....	3098
?_forward_trig_transform.....	3101
?_backward_trig_transform.....	3102
free_trig_transform.....	3104
Common Parameters.....	3105
Implementation Details.....	3108
Fast Poisson Solver Routines	3110
Poisson Solver Implementation.....	3110
Sequence of Invoking Poisson Solver Routines.....	3117
Interface Description.....	3119
Routines for the Cartesian Solver.....	3119
?_init_Helmholtz_2D/?_init_Helmholtz_3D.....	3119
_commit_Helmholtz_2D/?_commit_Helmholtz_3D.....	3122
?_Helmholtz_2D/?_Helmholtz_3D.....	3125
free_Helmholtz_2D/free_Helmholtz_3D.....	3128
Routines for the Spherical Solver.....	3128
?_init_sph_p/?_init_sph_np.....	3129
?_commit_sph_p/?_commit_sph_np.....	3130
?_sph_p/?_sph_np.....	3132
free_sph_p/free_sph_np.....	3134
Common Parameters.....	3135
ipar.....	3135
dpar and spar.....	3139
Caveat on Parameter Modifications.....	3141
Parameters That Define Boundary Conditions.....	3142
Implementation Details.....	3145
Calling PDE Support Routines from Fortran.....	3151

Chapter 13: Nonlinear Optimization Problem Solvers

Organization and Implementation.....	3153
Routine Naming Conventions.....	3154
Nonlinear Least Squares Problem without Constraints.....	3154
?trnlsp_init.....	3155
?trnlsp_check.....	3157
?trnlsp_solve.....	3159
?trnlsp_get.....	3161
?trnlsp_delete.....	3162
Nonlinear Least Squares Problem with Linear (Bound) Constraints.....	3163
?trnlspbc_init.....	3163
?trnlspbc_check.....	3165
?trnlspbc_solve.....	3168
?trnlspbc_get.....	3170
?trnlspbc_delete.....	3171
Jacobian Matrix Calculation Routines.....	3172
?jacobi_init.....	3172

?jacobi_solve.....	3173
?jacobi_delete.....	3175
?jacobi.....	3175
?jacobix.....	3177

Chapter 14: Support Functions

Using a Fortran Interface Module for Support Functions.....	3184
Version Information.....	3184
mkl_get_version.....	3184
mkl_get_version_string.....	3186
Threading Control.....	3187
mkl_set_num_threads.....	3188
mkl_domain_set_num_threads.....	3189
mkl_set_num_threads_local.....	3191
mkl_set_dynamic.....	3193
mkl_get_max_threads.....	3194
mkl_domain_get_max_threads.....	3195
mkl_get_dynamic.....	3196
Error Handling.....	3198
Error Handling for Linear Algebra Routines.....	3198
xerbla.....	3198
pxerbla.....	3199
Handling Fatal Errors.....	3200
mkl_set_exit_handler.....	3200
Character Equality Testing.....	3201
Isame.....	3202
Isamen.....	3202
Timing.....	3203
second/dsecnd.....	3203
mkl_get_cpu_clocks.....	3204
mkl_get_cpu_frequency.....	3205
mkl_get_max_cpu_frequency.....	3206
mkl_get_clocks_frequency.....	3206
Memory Management.....	3207
mkl_free_buffers.....	3207
mkl_thread_free_buffers.....	3208
mkl_disable_fast_mm.....	3209
mkl_mem_stat.....	3210
mkl_peak_mem_usage.....	3211
mkl_malloc.....	3212
mkl_calloc.....	3213
mkl_realloc.....	3214
mkl_free.....	3215
Usage Examples for the Memory Functions.....	3216
Single Dynamic Library Control.....	3219
mkl_set_interface_layer.....	3219
mkl_set_threading_layer.....	3220
mkl_set_xerbla.....	3221
mkl_set_progress.....	3222
Intel Many Integrated Core Architecture Support.....	3223

mkl_mic_enable.....	3224
mkl_mic_disable.....	3225
mkl_mic_get_device_count.....	3225
mkl_mic_set_workdivision.....	3226
mkl_mic_get_workdivision.....	3228
mkl_mic_set_max_memory.....	3229
mkl_mic_free_memory.....	3231
mkl_mic_register_memory.....	3232
mkl_mic_set_device_num_threads.....	3233
mkl_mic_set_resource_limit.....	3235
mkl_mic_get_resource_limit.....	3238
mkl_mic_set_offload_report.....	3239
Conditional Numerical Reproducibility Control.....	3239
mkl_cbwr_set.....	3240
mkl_cbwr_get.....	3241
mkl_cbwr_get_auto_branch.....	3242
Named Constants for CNR Control.....	3243
Reproducibility Conditions.....	3244
Usage Examples for CNR Support Functions.....	3245
Miscellaneous.....	3246
mkl_progress.....	3246
mkl_enable_instructions.....	3248
mkl_set_env_mode.....	3249
mkl_verbose.....	3250

Chapter 15: BLACS Routines

Matrix Shapes.....	3253
Repeatability and Coherence.....	3254
BLACS Combine Operations.....	3257
?gamx2d.....	3258
?gamm2d.....	3259
?gsum2d.....	3261
BLACS Point To Point Communication.....	3262
?gesd2d.....	3264
?trsd2d.....	3264
?gerv2d.....	3265
?trrv2d.....	3266
BLACS Broadcast Routines.....	3266
?gebs2d.....	3268
?trbs2d.....	3268
?gebr2d.....	3269
?trbr2d.....	3270
BLACS Support Routines.....	3271
Initialization Routines.....	3271
blacs_pinfo.....	3271
blacs_setup.....	3272
blacs_get.....	3272
blacs_set.....	3273
blacs_gridinit.....	3275
blacs_gridmap.....	3276

Destruction Routines.....	3277
blacs_freebuff.....	3278
blacs_gridexit.....	3278
blacs_abort.....	3278
blacs_exit.....	3279
Informational Routines.....	3279
blacs_gridinfo.....	3280
blacs_pnum.....	3280
blacs_pcoord.....	3280
Miscellaneous Routines.....	3281
blacs_barrier.....	3281
Examples of BLACS Routines Usage.....	3282

Chapter 16: Data Fitting Functions

Naming Conventions.....	3293
Data Types.....	3294
Mathematical Conventions for Data Fitting Functions.....	3294
Data Fitting Usage Model.....	3297
Data Fitting Usage Examples.....	3297
Task Status and Error Reporting.....	3303
Task Creation and Initialization Routines.....	3305
df?newtask1d.....	3305
Task Configuration Routines.....	3307
df?editppspline1d.....	3308
df?editptr.....	3315
dfieditval.....	3316
df?editidxptr.....	3318
df?queryptr.....	3320
dfiqueryval.....	3321
df?queryidxptr.....	3322
Computational Routines.....	3323
df?construct1d.....	3324
df?interpolate1d/df?interpolateex1d.....	3325
df?integrate1d/df?integrateex1d.....	3332
df?searchcells1d/df?searchcellsex1d.....	3337
df?interpcallback.....	3340
df?integrcallback.....	3341
df?searchcellscallback.....	3343
Task Destructors.....	3345
dfdeletetask.....	3345

Appendix A: Linear Solvers Basics

Sparse Linear Systems.....	3347
Matrix Fundamentals.....	3347
Direct Method.....	3348
Sparse Matrix Storage Formats.....	3352

Appendix B: Routine and Function Arguments

Vector Arguments in BLAS.....	3365
Vector Arguments in VML.....	3366
Matrix Arguments.....	3366

Appendix C: Specific Features of Fortran 95 Interfaces for LAPACK Routines

Interfaces Identical to Netlib.....	3373
Interfaces with Replaced Argument Names.....	3374
Modified Netlib Interfaces.....	3376
Interfaces Absent From Netlib.....	3377
Interfaces of New Functionality.....	3380

Appendix D: FFTW Interface to Intel® Math Kernel Library

Notational Conventions	3381
FFTW2 Interface to Intel® Math Kernel Library	3381
Wrappers Reference.....	3381
One-dimensional Complex-to-complex FFTs	3381
Multi-dimensional Complex-to-complex FFTs.....	3382
One-dimensional Real-to-half-complex/Half-complex-to-real FFTs.....	3382
Multi-dimensional Real-to-complex/Complex-to-real FFTs.....	3382
Multi-threaded FFTW.....	3383
FFTW Support Functions.....	3383
Calling Wrappers from Fortran.....	3383
Limitations of the FFTW2 Interface to Intel MKL.....	3385
Installation.....	3385
Creating the Wrapper Library.....	3385
Application Assembling	3386
Running Examples	3386
MPI FFTW Wrappers.....	3386
MPI FFTW Wrappers Reference.....	3387
Creating MPI FFTW Wrapper Library.....	3388
Application Assembling with MPI FFTW Wrapper Library.....	3388
Running Examples	3389
FFTW3 Interface to Intel® Math Kernel Library.....	3389
Using FFTW3 Wrappers.....	3390
Calling Wrappers from Fortran.....	3391
Building Your Own Wrapper Library.....	3392
Building an Application.....	3392
Running Examples	3393
MPI FFTW Wrappers.....	3393
Building Your Own Wrapper Library.....	3393
Building an Application.....	3394
Running Examples.....	3394

Appendix E: Code Examples

BLAS Code Examples.....	3395
Fourier Transform Functions Code Examples.....	3398
FFT Code Examples.....	3398
Examples of Using Multi-Threading for FFT Computation.....	3406
Examples for Cluster FFT Functions.....	3409
Auxiliary Data Transformations.....	3411

Bibliography

Glossary

Legal Information

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Cilk, Intel, the Intel logo, Intel Atom, Intel Core, Intel Inside, Intel NetBurst, Intel SpeedStep, Intel vPro, Intel Xeon Phi, Intel XScale, Itanium, MMX, Pentium, Thunderbolt, Ultrabook, VTune and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java is a registered trademark of Oracle and/or its affiliates.

Third Party Content

Intel® Math Kernel Library (Intel® MKL) includes content from several 3rd party sources that was originally governed by the licenses referenced below:

- Portions® Copyright 2001 Hewlett-Packard Development Company, L.P.
- Sections on the Linear Algebra PACKage (LAPACK) routines include derivative work portions that have been copyrighted:

© 1991, 1992, and 1998 by The Numerical Algorithms Group, Ltd.

- Intel MKL supports LAPACK 3.5 set of computational, driver, auxiliary and utility routines under the following license:

Copyright © 1992-2011 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.

Copyright © 2000-2011 The University of California Berkeley. All rights reserved.

Copyright © 2006-2012 The University of Colorado Denver. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The original versions of LAPACK from which that part of Intel MKL was derived can be obtained from <http://www.netlib.org/lapack/index.html>. The authors of LAPACK are E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.

- The original versions of the Basic Linear Algebra Subprograms (BLAS) from which the respective part of Intel® MKL was derived can be obtained from <http://www.netlib.orgblas/index.html>.
- XBLAS is distributed under the following copyright:

Copyright © 2008-2009 The University of California Berkeley. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- The original versions of the Basic Linear Algebra Communication Subprograms (BLACS) from which the respective part of Intel MKL was derived can be obtained from <http://www.netlib.org/blacs/index.html>. The authors of BLACS are Jack Dongarra and R. Clint Whaley.
- The original versions of Scalable LAPACK (ScaLAPACK) from which the respective part of Intel® MKL was derived can be obtained from <http://www.netlib.org/scalapack/index.html>. The authors of ScaLAPACK are L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley.
- The original versions of the Parallel Basic Linear Algebra Subprograms (PBLAS) routines from which the respective part of Intel® MKL was derived can be obtained from http://www.netlib.org/scalapack/html/pblas_qref.html.
- PARDISO (PARallel DIrect SOLver)* in Intel® MKL was originally developed by the Department of Computer Science at the University of Basel (<http://www.unibas.ch>). It can be obtained at <http://www.pardiso-project.org>.
- The Extended Eigensolver functionality is based on the Feast solver package and is distributed under the following license:

Copyright © 2009, The Regents of the University of Massachusetts, Amherst.
Developed by E. Polizzi
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Some Fast Fourier Transform (FFT) functions in this release of Intel® MKL have been generated by the SPIRAL software generation system (<http://www.spiral.net/>) under license from Carnegie Mellon University. The authors of SPIRAL are Markus Puschel, Jose Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo.
- Open MPI is distributed under the New BSD license, listed below.

Most files in this release are marked with the copyrights of the organizations who have edited them. The copyrights below are in no particular order and generally reflect members of the Open MPI core team who have contributed code to this release. The copyrights for code used under license from other parties are included in the corresponding files.

Copyright © 2004-2010 The Trustees of Indiana University and Indiana University Research and Technology Corporation. All rights reserved.

Copyright © 2004-2010 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.

Copyright © 2004-2010 High Performance Computing Center Stuttgart, University of Stuttgart. All rights reserved.

Copyright © 2004-2008 The Regents of the University of California. All rights reserved.

Copyright © 2006-2010 Los Alamos National Security, LLC. All rights reserved.

Copyright © 2006-2010 Cisco Systems, Inc. All rights reserved.

Copyright © 2006-2010 Voltaire, Inc. All rights reserved.

Copyright © 2006-2011 Sandia National Laboratories. All rights reserved.

Copyright © 2006-2010 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.

Copyright © 2006-2010 The University of Houston. All rights reserved.

Copyright © 2006-2009 Myricom, Inc. All rights reserved.

Copyright © 2007-2008 UT-Battelle, LLC. All rights reserved.

Copyright © 2007-2010 IBM Corporation. All rights reserved.

Copyright © 1998-2005 Forschungszentrum Juelich, Juelich Supercomputing Centre, Federal Republic of Germany

Copyright © 2005-2008 ZIH, TU Dresden, Federal Republic of Germany

Copyright © 2007 Evergrid, Inc. All rights reserved.

Copyright © 2008 Chelsio, Inc. All rights reserved.

Copyright © 2008-2009 Institut National de Recherche en Informatique. All rights reserved.

Copyright © 2007 Lawrence Livermore National Security, LLC. All rights reserved.

Copyright © 2007-2009 Mellanox Technologies. All rights reserved.

Copyright © 2006-2010 QLogic Corporation. All rights reserved.

Copyright © 2008-2010 Oak Ridge National Labs. All rights reserved.

Copyright © 2006-2010 Oracle and/or its affiliates. All rights reserved.

Copyright © 2009 Bull SAS. All rights reserved.

Copyright © 2010 ARM Ltd. All rights reserved.

Copyright © 2010-2011 Alex Brick . All rights reserved.

Copyright © 2012 The University of Wisconsin-La Crosse. All rights reserved.

Copyright © 2013-2014 Intel, Inc. All rights reserved.

Copyright © 2011-2014 NVIDIA Corporation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Copyright© 2015, Intel Corporation. All rights reserved.

Introducing the Intel® Math Kernel Library

The Intel® Math Kernel Library (Intel® MKL) improves performance with math routines for software applications that solve large computational problems. Intel MKL provides BLAS and LAPACK linear algebra routines, fast Fourier transforms, vectorized math functions, random number generation functions, and other functionality.

Intel MKL is optimized for the latest Intel processors, including processors with multiple cores (see the *Intel MKL Release Notes* for the full list of supported processors). Intel MKL also performs well on non-Intel processors.

For more details about functionality provided by Intel MKL, see the [Function Domains](#) section.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Getting Help and Support

Intel MKL provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, check: <http://www.intel.com/software/products/support>.

Intel also provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more (visit <http://www.intel.com/software/products/>).

Registering your product entitles you to one year of technical support and product updates through Intel® Premier Support. Intel Premier Support is an interactive issue management and communication web site providing these services:

- Submit issues and review their status.
- Download product updates anytime of the day.

To register your product, contact Intel, or seek product support, please visit <http://www.intel.com/software/products/support>.

What's New

This Reference Manual documents Intel® Math Kernel Library (Intel® MKL) 11.2 update 2 release.

The manual has been updated to reflect enhancements to the product, and minor improvements have been made.

Additionally, minor updates have been made to correct errors in the manual.

Notational Conventions

This manual uses the following terms to refer to operating systems:

Windows* OS	This term refers to information that is valid on all supported Windows* operating systems.
Linux* OS	This term refers to information that is valid on all supported Linux* operating systems.
OS X*	This term refers to information that is valid on Intel®-based systems running the OS X* operating system.

This manual uses the following notational conventions:

- Routine name shorthand (for example, `?ungqr` instead of `cungqr/zungqr`).
- Font conventions used for distinction between the text and the code.

Routine Name Shorthand

For shorthand, names that contain a question mark "?" represent groups of routines with similar functionality. Each group typically consists of routines used with four basic data types: single-precision real, double-precision real, single-precision complex, and double-precision complex. The question mark is used to indicate any or all possible varieties of a function; for example:

`?swap` Refers to all four data types of the vector-vector `?swap` routine:
`sswap`, `dswap`, `cswap`, and `zswap`.

Font Conventions

The following font conventions are used:

UPPERCASE COURIER	Data type used in the description of input and output parameters for Fortran interface. For example, <code>CHARACTER*1</code> .
lowercase courier	Code examples: <code>a(k+i,j) = matrix(i,j)</code> and data types for C interface, for example, <code>const float*</code>
lowercase courier mixed with uppercase courier	Function names for C interface; for example, <code>vmlSetMode</code>
lowercase courier italic	Variables in arguments and parameters description. For example, <code>incx</code> .
*	Used as a multiplication symbol in code examples and equations and where required by the programming language syntax.

Function Domains

The Intel® Math Kernel Library includes Fortran routines and functions optimized for Intel® processor-based computers running operating systems that support multiprocessing. In addition to the Fortran interface, Intel MKL includes a C-language interface for the Discrete Fourier transform functions, as well as for the Vector Mathematical Library and Vector Statistical Library functions. For hardware and software requirements to use Intel MKL, see *Intel® MKL Release Notes*.

The Intel® Math Kernel Library includes the following groups of routines:

- Basic Linear Algebra Subprograms (BLAS):
 - vector operations
 - matrix-vector operations
 - matrix-matrix operations
- Sparse BLAS Level 1, 2, and 3 (basic operations on sparse vectors and matrices)
- LAPACK routines for solving systems of linear equations
- LAPACK routines for solving least squares problems, eigenvalue and singular value problems, and Sylvester's equations
- Auxiliary, utility, and test LAPACK routines
- ScaLAPACK computational, driver and auxiliary routines (only in Intel MKL for Linux* and Windows* operating systems)
- PBLAS routines for distributed vector, matrix-vector, and matrix-matrix operation
- Direct and Iterative Sparse Solver routines, including a solver based on the PARDISO* sparse solver and the Intel MKL Parallel Direct Sparse Solver for Clusters
- Direct Sparse Solver (DSS)
- Extended Eigensolver routines for solving symmetric standard or generalized symmetric definite eigenvalue problems using the Feast algorithm
- Vector Mathematical Library (VML) functions for computing core mathematical functions on vector arguments (with Fortran and C interfaces)
- Vector Statistical Library (VSL) functions for generating vectors of pseudorandom numbers with different types of statistical distributions and for performing convolution and correlation computations
- General Fast Fourier Transform (FFT) Functions, providing fast computation of Discrete Fourier Transform via the FFT algorithms and having Fortran and C interfaces
- Cluster FFT functions (only in Intel MKL for Linux* and Windows* operating systems)
- Tools for solving partial differential equations - trigonometric transform routines and Poisson solver
- Optimization Solver routines for solving nonlinear least squares problems through the Trust-Region (TR) algorithms and computing Jacobi matrix by central differences
- Basic Linear Algebra Communication Subprograms (BLACS) that are used to support a linear algebra oriented message passing interface
- Data Fitting functions for spline-based approximation of functions, derivatives and integrals of functions, and search

For specific issues on using the library, also see the *Intel® MKL Release Notes*.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

BLAS Routines

The BLAS routines and functions are divided into the following groups according to the operations they perform:

- [BLAS Level 1 Routines](#) perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
- [BLAS Level 2 Routines](#) perform matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.
- [BLAS Level 3 Routines](#) perform matrix-matrix operations, such as matrix-matrix multiplication, rank-k update, and solution of triangular systems.

Starting from release 8.0, Intel® MKL also supports the Fortran 95 interface to the BLAS routines.

Starting from release 10.1, a number of [BLAS-like Extensions](#) are added to enable the user to perform certain data manipulation, including matrix in-place and out-of-place transposition operations combined with simple matrix arithmetic operations.

Sparse BLAS Routines

The [Sparse BLAS Level 1 Routines and Functions](#) and [Sparse BLAS Level 2 and Level 3 Routines](#) routines and functions operate on sparse vectors and matrices. These routines perform vector operations similar to the BLAS Level 1, 2, and 3 routines. The Sparse BLAS routines take advantage of vector and matrix sparsity: they allow you to store only non-zero elements of vectors and matrices. Intel MKL also supports Fortran 95 interface to Sparse BLAS routines.

LAPACK Routines

The Intel® Math Kernel Library fully supports LAPACK 3.5 set of computational, driver, auxiliary and utility routines.

The original versions of LAPACK from which that part of Intel MKL was derived can be obtained from <http://www.netlib.org/lapack/index.html>. The authors of LAPACK are E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.

The LAPACK routines can be divided into the following groups according to the operations they perform:

- Routines for solving systems of linear equations, factoring and inverting matrices, and estimating condition numbers (see [LAPACK Routines: Linear Equations](#)).
- Routines for solving least squares problems, eigenvalue and singular value problems, and Sylvester's equations (see [LAPACK Routines: Least Squares and Eigenvalue Problems](#)).
- Auxiliary and utility routines used to perform certain subtasks, common low-level computation or related tasks (see [LAPACK Auxiliary Routines](#) and [LAPACK Utility Functions and Routines](#)).

Starting from release 8.0, Intel MKL also supports the Fortran 95 interface to LAPACK computational and driver routines. This interface provides an opportunity for simplified calls of LAPACK routines with fewer required arguments.

ScaLAPACK Routines

The ScaLAPACK package (included only with the Intel® MKL versions for Linux* and Windows* operating systems, see [Chapter 6](#) and [Chapter 7](#)) runs on distributed-memory architectures and includes routines for solving systems of linear equations, solving linear least squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

The original versions of ScaLAPACK from which that part of Intel MKL was derived can be obtained from <http://www.netlib.org/scalapack/index.html>. The authors of ScaLAPACK are L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley.

The Intel MKL version of ScaLAPACK is optimized for Intel® processors and uses MPICH version of MPI as well as Intel MPI.

PBLAS Routines

The PBLAS routines perform operations with distributed vectors and matrices.

- [PBLAS Level 1 Routines](#) perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
- [PBLAS Level 2 Routines](#) perform distributed matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.
- [PBLAS Level 3 Routines](#) perform distributed matrix-matrix operations, such as matrix-matrix multiplication, rank- k update, and solution of triangular systems.

Intel MKL provides the PBLAS routines with interface similar to the interface used in the Netlib PBLAS (part of the ScaLAPACK package, see http://www.netlib.org/scalapack/html/pblas_qref.html).

Sparse Solver Routines

Direct sparse solver routines in Intel MKL (see [Chapter 8](#)) solve symmetric and symmetrically-structured sparse matrices with real or complex coefficients. For symmetric matrices, these Intel MKL subroutines can solve both positive-definite and indefinite systems. Intel MKL includes a solver based on the PARDISO* sparse solver, referred to as Intel MKL PARDISO, as well as an alternative set of user callable direct sparse solver routines.

If you use the Intel MKL PARDISO sparse solver, please cite:

O.Schenk and K.Gartner. Solving unsymmetric sparse systems of linear equations with PARDISO. *J. of Future Generation Computer Systems*, 20(3):475-487, 2004.

Intel MKL provides also an iterative sparse solver (see [Chapter 8](#)) that uses Sparse BLAS level 2 and 3 routines and works with different sparse data formats.

Extended Eigensolver Routines

The [Extended Eigensolver functionality](#) is a set of high-performance numerical routines for solving standard ($Ax = \lambda x$) and generalized ($Ax = \lambda Bx$) eigenvalue problems, where A and B are symmetric or Hermitian. It yields all the eigenvalues and eigenvectors within a given search interval. It is based on the Feast algorithm, an innovative fast and stable numerical algorithm presented in [\[Polizzi09\]](#), which deviates fundamentally from the traditional Krylov subspace iteration based techniques (Arnoldi and Lanczos algorithms [\[Bai00\]](#)) or other Davidson-Jacobi techniques [\[Sleijpen96\]](#). The Feast algorithm is inspired by the density-matrix representation and contour integration technique in quantum mechanics.

It is free from orthogonalization procedures. Its main computational tasks consist of solving very few inner independent linear systems with multiple right-hand sides and one reduced eigenvalue problem orders of magnitude smaller than the original one. The Feast algorithm combines simplicity and efficiency and offers many important capabilities for achieving high performance, robustness, accuracy, and scalability on parallel architectures. This algorithm is expected to significantly augment numerical performance in large-scale modern applications.

Some of the characteristics of the Feast algorithm [\[Polizzi09\]](#) are:

- Converges quickly in 2-3 iterations with very high accuracy
- Naturally captures all eigenvalue multiplicities
- No explicit orthogonalization procedure
- Can reuse the basis of pre-computed subspace as suitable initial guess for performing outer-refinement iterations

This capability can also be used for solving a series of eigenvalue problems that are close one another.

- The number of internal iterations is independent of the size of the system and the number of eigenpairs in the search interval
- The inner linear systems can be solved either iteratively (even with modest relative residual error) or directly

VML Functions

The Vector Mathematical Library (VML) functions (see [Chapter 9](#)) include a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic, etc.) that operate on vectors of real and complex numbers.

Application programs that might significantly improve performance with VML include nonlinear programming software, integrals computation, and many others. VML provides interfaces both for Fortran and C languages.

Statistical Functions

The Vector Statistical Library (VSL) contains three sets of functions (see [Chapter 10](#)) providing:

- Pseudorandom, quasi-random, and non-deterministic random number generator subroutines implementing basic continuous and discrete distributions. To provide best performance, the VSL subroutines use calls to highly optimized Basic Random Number Generators (BRNGs) and a library of vector mathematical functions.
- A wide variety of convolution and correlation operations.
- Initial statistical analysis of raw single and double precision multi-dimensional datasets.

Fourier Transform Functions

The Intel® MKL multidimensional Fast Fourier Transform (FFT) functions with mixed radix support (see [Chapter 11](#)) provide uniformity of discrete Fourier transform computation and combine functionality with ease of use. Both Fortran and C interface specification are given. There is also a cluster version of FFT functions, which runs on distributed-memory architectures and is provided only in Intel MKL versions for the Linux* and Windows* operating systems.

The FFT functions provide fast computation via the FFT algorithms for arbitrary lengths. See the *Intel® MKL User's Guide* for the specific radices supported.

Partial Differential Equations Support

Intel® MKL provides tools for solving Partial Differential Equations (PDE) (see [Chapter 13](#)). These tools are Trigonometric Transform interface routines and Poisson Solver.

The Trigonometric Transform routines may be helpful to users who implement their own solvers similar to the Intel MKL Poisson Solver. The users can improve performance of their solvers by using fast sine, cosine, and staggered cosine transforms implemented in the Trigonometric Transform interface.

The Poisson Solver is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The Trigonometric Transform interface, which underlies the solver, is based on the Intel MKL FFT interface (refer to [Chapter 11](#)), optimized for Intel® processors.

Nonlinear Optimization Problem Solvers

Intel® MKL provides Nonlinear Optimization Problem Solver routines (see [Chapter 14](#)) that can be used to solve nonlinear least squares problems with or without linear (bound) constraints through the Trust-Region (TR) algorithms and compute Jacobi matrix by central differences.

Support Functions

The Intel® MKL support functions (see [Chapter 15](#)) are used to support the operation of the Intel MKL software and provide basic information on the library and library operation, such as the current library version, timing, setting and measuring of CPU frequency, error handling, and memory allocation.

Starting from release 10.0, the Intel MKL support functions provide additional threading control.

Starting from release 10.1, Intel MKL selectively supports a *Progress Routine* feature to track progress of a lengthy computation and/or interrupt the computation using a callback function mechanism. The user application can define a function called `mkl_progress` that is regularly called from the Intel MKL routine supporting the progress routine feature. See [the Progress Routines](#) section in [Chapter 15](#) for reference. Refer to a specific LAPACK or DSS/PARDISO function description to see whether the function supports this feature or not.

BLACS Routines

The Intel® Math Kernel Library implements routines from the BLACS (Basic Linear Algebra Communication Subprograms) package (see [Chapter 16](#)) that are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The original versions of BLACS from which that part of Intel MKL was derived can be obtained from <http://www.netlib.org/blacs/index.html>. The authors of BLACS are Jack Dongarra and R. Clint Whaley.

Data Fitting Functions

The Data Fitting component includes a set of highly-optimized implementations of algorithms for the following spline-based computations:

- spline construction
- interpolation including computation of derivatives and integration
- search

The algorithms operate on single and double vector-valued functions set in the points of the given partition. You can use Data Fitting algorithms in applications that are based on data approximation. See [Data Fitting Functions](#) for more information.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Performance Enhancements

The Intel® Math Kernel Library has been optimized by exploiting both processor and system features and capabilities. Special care has been given to those routines that most profit from cache-management techniques. These especially include matrix-matrix operation routines such as `dgemm()`.

In addition, code optimization techniques have been applied to minimize dependencies of scheduling integer and floating-point units on the results within the processor.

The major optimization techniques used throughout the library include:

- Loop unrolling to minimize loop management costs

- Blocking of data to improve data reuse opportunities
- Copying to reduce chances of data eviction from cache
- Data prefetching to help hide memory latency
- Multiple simultaneous operations (for example, dot products in `dgemm`) to eliminate stalls due to arithmetic unit pipelines
- Use of hardware features such as the SIMD arithmetic units, where appropriate

These are techniques from which the arithmetic code benefits the most.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Parallelism

In addition to the performance enhancements discussed above, Intel® MKL offers performance gains through parallelism provided by the symmetric multiprocessing performance (SMP) feature. You can obtain improvements from SMP in the following ways:

- One way is based on user-managed threads in the program and further distribution of the operations over the threads based on data decomposition, domain decomposition, control decomposition, or some other parallelizing technique. Each thread can use any of the Intel MKL functions (except for the deprecated `?lacon` LAPACK routine) because the library has been designed to be thread-safe.
- Another method is to use the FFT and BLAS level 3 routines. They have been parallelized and require no alterations of your application to gain the performance enhancements of multiprocessing. Performance using multiple processors on the level 3 BLAS shows excellent scaling. Since the threads are called and managed within the library, the application does not need to be recompiled thread-safe (see also [Fortran 95 Interface Conventions](#) in Chapter 2).
- Yet another method is to use *tuned LAPACK routines*. Currently these include the single- and double precision flavors of routines for QR factorization of general matrices, triangular factorization of general and symmetric positive-definite matrices, solving systems of equations with such matrices, as well as solving symmetric eigenvalue problems.

For instructions on setting the number of available processors for the BLAS level 3 and LAPACK routines, see [Intel® MKL User's Guide](#).

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

C Datatypes Specific to Intel MKL

The `mkl_types.h` file defines datatypes specific to Intel MKL.

C/C++ Type	Fortran Type	LP32 Equivalent (Size in Bytes)	LP64 Equivalent (Size in Bytes)	ILP64 Equivalent (Size in Bytes)
<code>MKL_INT</code> (MKL integer)	<code>INTEGER</code> (default <code>INTEGER</code>)	C/C++: <code>int</code> Fortran: <code>INTEGER*4</code> (4 bytes)	C/C++: <code>int</code> Fortran: <code>INTEGER*4</code> (4 bytes)	C/C++: <code>long long</code> (or define <code>MKL_ILP64</code> macros) Fortran: <code>INTEGER*8</code> (8 bytes)
<code>MKL_UINT</code> (MKL unsigned integer)	N/A	C/C++: <code>unsigned int</code> (4 bytes)	C/C++: <code>unsigned int</code> (4 bytes)	C/C++: <code>unsigned long long</code> (8 bytes)
<code>MKL_LONG</code> (MKL long integer)	N/A	C/C++: <code>long</code> (4 bytes)	C/C++: <code>long</code> (Windows: 4 bytes) (Linux, Mac: 8 bytes)	C/C++: <code>long</code> (8 bytes)
<code>MKL_Complex8</code> (Like C99 complex float)	<code>COMPLEX*8</code>	(8 bytes)	(8 bytes)	(8 bytes)
<code>MKL_Complex16</code> (Like C99 complex double)	<code>COMPLEX*16</code>	(16 bytes)	(16 bytes)	(16 bytes)

You can redefine datatypes specific to Intel MKL. One reason to do this is if you have your own types which are binary-compatible with Intel MKL datatypes, with the same representation or memory layout. To redefine a datatype, use one of these methods:

- Insert the `#define` statement redefining the datatype before the `mkl.h` header file `#include` statement. For example,

```
#define MKL_INT size_t
#include "mkl.h"
```

- Use the compiler `-D` option to redefine the datatype. For example,

```
...-DMKL_INT=size_t...
```

NOTE

As the user, if you redefine Intel MKL datatypes you are responsible for making sure that your definition is compatible with that of Intel MKL. If not, it might cause unpredictable results or crash the application.

BLAS and Sparse BLAS Routines

This chapter describes the Intel® Math Kernel Library implementation of the BLAS and Sparse BLAS routines, and BLAS-like extensions. The routine descriptions are arranged in several sections:

- [BLAS Level 1 Routines](#) (vector-vector operations)
- [BLAS Level 2 Routines](#) (matrix-vector operations)
- [BLAS Level 3 Routines](#) (matrix-matrix operations)
- [Sparse BLAS Level 1 Routines](#) (vector-vector operations).
- [Sparse BLAS Level 2 and Level 3 Routines](#) (matrix-vector and matrix-matrix operations)
- [BLAS-like Extensions](#)

Each section presents the routine and function group descriptions in alphabetical order by routine or function group name; for example, the `?asum` group, the `?axpy` group. The question mark in the group name corresponds to different character codes indicating the data type (`s`, `d`, `c`, and `z` or their combination); see [Routine Naming Conventions](#).

When BLAS or Sparse BLAS routines encounter an error, they call the error reporting routine `xerbla`.

In BLAS Level 1 groups `i?amax` and `i?amin`, an "i" is placed before the data-type indicator and corresponds to the index of an element in the vector. These groups are placed in the end of the BLAS Level 1 section.

BLAS Routines

Routine Naming Conventions

BLAS routine names have the following structure:

`<character> <name> <mod> ()`

The `<character>` field indicates the data type:

<code>s</code>	real, single precision
<code>c</code>	complex, single precision
<code>d</code>	real, double precision
<code>z</code>	complex, double precision

Some routines and functions can have combined character codes, such as `sc` or `dz`.

For example, the function `scasum` uses a complex input array and returns a real value.

The `<name>` field, in BLAS level 1, indicates the operation type. For example, the BLAS level 1 routines `?dot`, `?rot`, `?swap` compute a vector dot product, vector rotation, and vector swap, respectively.

In BLAS level 2 and 3, `<name>` reflects the matrix argument type:

<code>ge</code>	general matrix
<code>gb</code>	general band matrix
<code>sy</code>	symmetric matrix

sp	symmetric matrix (packed storage)
sb	symmetric band matrix
he	Hermitian matrix
hp	Hermitian matrix (packed storage)
hb	Hermitian band matrix
tr	triangular matrix
tp	triangular matrix (packed storage)
tb	triangular band matrix.

The $\langle mod \rangle$ field, if present, provides additional details of the operation. BLAS level 1 names can have the following characters in the $\langle mod \rangle$ field:

c	conjugated vector
u	unconjugated vector
g	Givens rotation construction
m	modified Givens rotation
mg	modified Givens rotation construction

BLAS level 2 names can have the following characters in the $\langle mod \rangle$ field:

mv	matrix-vector product
sv	solving a system of linear equations with a single unknown vector
r	rank-1 update of a matrix
r2	rank-2 update of a matrix.

BLAS level 3 names can have the following characters in the $\langle mod \rangle$ field:

mm	matrix-matrix product
sm	solving a system of linear equations with multiple unknown vectors
rk	rank- k update of a matrix
r2k	rank- $2k$ update of a matrix.

The examples below illustrate how to interpret BLAS routine names:

ddot	$\langle d \rangle \langle dot \rangle$: double-precision real vector-vector dot product
cdotc	$\langle c \rangle \langle dot \rangle \langle c \rangle$: complex vector-vector dot product, conjugated
scasum	$\langle sc \rangle \langle asum \rangle$: sum of magnitudes of vector elements, single precision real output and single precision complex input
cdotu	$\langle c \rangle \langle dot \rangle \langle u \rangle$: vector-vector dot product, unconjugated, complex
sgemv	$\langle s \rangle \langle ge \rangle \langle mv \rangle$: matrix-vector product, general matrix, single precision

ztrmm	<code><z> <tr> <mm></code> : matrix-matrix product, triangular matrix, double-precision complex.
-------	--

Sparse BLAS level 1 naming conventions are similar to those of BLAS level 1. For more information, see [Naming Conventions](#).

Fortran 95 Interface Conventions

Fortran 95 interface to BLAS and Sparse BLAS Level 1 routines is implemented through wrappers that call respective FORTRAN 77 routines. This interface uses such features of Fortran 95 as assumed-shape arrays and optional arguments to provide simplified calls to BLAS and Sparse BLAS Level 1 routines with fewer parameters.

NOTE

For BLAS, Intel MKL offers two types of Fortran 95 interfaces:

- using `mkl blas.fi` only through `include 'mkl.fi'` statement. Such interfaces allow you to make use of the original LAPACK routines with all their arguments
 - using `blas.f90` that includes improved interfaces. This file is used to generate the module files `blas95.mod` and `f95_precision.mod`. See also section "Fortran 95 interfaces and wrappers to LAPACK and BLAS" of *Intel® MKL User's Guide* for details. The module files are used to process the FORTRAN use clauses referencing the BLAS interface: `use blas95` and `use f95_precision`.
-

The main conventions used in Fortran 95 interface are as follows:

- The names of parameters used in Fortran 95 interface are typically the same as those used for the respective generic (FORTRAN 77) interface. In rare cases formal argument names may be different.
- Some input parameters such as array dimensions are not required in Fortran 95 and are skipped from the calling sequence. Array dimensions are reconstructed from the user data that must exactly follow the required array shape.
- A parameter can be skipped if its value is completely defined by the presence or absence of another parameter in the calling sequence, and the restored value is the only meaningful value for the skipped parameter.
- Parameters specifying the increment values `incx` and `incy` are skipped. In most cases their values are equal to 1. In Fortran 95 an increment with different value can be directly established in the corresponding parameter.
- Some generic parameters are declared as optional in Fortran 95 interface and may or may not be present in the calling sequence. A parameter can be declared optional if it satisfies one of the following conditions:
 1. It can take only a few possible values. The default value of such parameter typically is the first value in the list; all exceptions to this rule are explicitly stated in the routine description.
 2. It has a natural default value.

Optional parameters are given in square brackets in Fortran 95 call syntax.

The particular rules used for reconstructing the values of omitted optional parameters are specific for each routine and are detailed in the respective "Fortran 95 Notes" subsection at the end of routine specification section. If this subsection is omitted, the Fortran 95 interface for the given routine does not differ from the corresponding FORTRAN 77 interface.

Note that this interface is not implemented in the current version of Sparse BLAS Level 2 and Level 3 routines.

C Interface Conventions

CBLAS, the C interface to the Basic Linear Algebra Subprograms (BLAS), provides a C language interface to BLAS routines for Intel MKL. While you can call the Fortran implementation of BLAS, for coding in C the CBLAS interface has some advantages such as allowing you to specify column-major or row-major ordering with the `layout` parameter.

For more information about calling Fortran routines from C in general, and specifically about calling BLAS and CBLAS routines, see " Mixed-language Programming with the Intel Math Kernel Library" in the *Intel Math Kernel Library User's Guide*.

In CBLAS, the Fortran routine names are prefixed with `cblas_` (for example, `dasum` becomes `cblas_dasum`). Names of all CBLAS functions are in lowercase letters.

Complex functions `?dotc` and `?dotu` become CBLAS subroutines (void functions); they return the complex result via a void pointer, added as the last parameter. CBLAS names of these functions are suffixed with `_sub`. For example, the BLAS function `cdotc` corresponds to `cblas_cdotc_sub`.

WARNING

Users of the CBLAS interface should be aware that the CBLAS are just a C interface to the BLAS, which is based on the FORTRAN standard and subject to the FORTRAN standard restrictions. In particular, the output parameters should not be referenced through more than one argument.

NOTE

This interface is not implemented in the Sparse BLAS Level 2 and Level 3 routines.

The arguments of CBLAS functions comply with the following rules:

- Input arguments are declared with the `const` modifier.
- Non-complex scalar input arguments are passed by value.
- Complex scalar input arguments are passed as void pointers.
- Array arguments are passed by address.
- BLAS character arguments are replaced by the appropriate enumerated type.
- Level 2 and Level 3 routines acquire an additional parameter of type `CBLAS_LAYOUT` as their first argument. This parameter specifies whether two-dimensional arrays are row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

Enumerated Types

The CBLAS interface uses the following enumerated types:

```
enum CBLAS_LAYOUT {
    CblasRowMajor=101, /* row-major arrays */
    CblasColMajor=102}; /* column-major arrays */

enum CBLAS_TRANSPOSE {
    CblasNoTrans=111, /* trans='N' */
    CblasTrans=112, /* trans='T' */
    CblasConjTrans=113}; /* trans='C' */

enum CBLAS_UPLO {
    CblasUpper=121, /* uplo ='U' */
    CblasLower=122}; /* uplo ='L' */

enum CBLAS_DIAG {
    CblasNonUnit=131, /* diag ='N' */
    CblasUnit=132}; /* diag ='U' */

enum CBLAS_SIDE {
    CblasLeft=141, /* side ='L' */
    CblasRight=142}; /* side ='R' */
```

Matrix Storage Schemes

Matrix arguments of BLAS routines can use the following storage schemes:

- *Full storage*: a matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} stored in the array element $a(i, j)$.

- *Packed storage* scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- *Band storage*: a band matrix is stored compactly in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.

For more information on matrix storage schemes, see [Matrix Arguments](#) in Appendix B.

Row-Major and Column-Major Ordering

The BLAS routines follow the Fortran convention of storing two-dimensional arrays using column-major ordering. When calling BLAS routines from C, remember that they require arrays to be in column-major format, not the row-major format that is the convention for C.

The CBLAS interface allows you to specify either column-major or row-major ordering for BLAS Level 2 and Level 3 routines, by setting the `layout` parameter to `CblasColMajor` or `CblasRowMajor`.

BLAS Level 1 Routines and Functions

BLAS Level 1 includes routines and functions, which perform vector-vector operations. [Table “BLAS Level 1 Routine Groups and Their Data Types”](#) lists the BLAS Level 1 routine and function groups and the data types associated with them.

BLAS Level 1 Routine and Function Groups and Their Data Types

Routine or Function Group	Data Types	Description
?asum	s, d, sc, dz	Sum of vector magnitudes (functions)
?axpy	s, d, c, z	Scalar-vector product (routines)
?copy	s, d, c, z	Copy vector (routines)
?dot	s, d	Dot product (functions)
?sdot	sd, d	Dot product with double precision (functions)
?dotc	c, z	Dot product conjugated (functions)
?dotu	c, z	Dot product unconjugated (functions)
?nrm2	s, d, sc, dz	Vector 2-norm (Euclidean norm) (functions)
?rot	s, d, cs, zd	Plane rotation of points (routines)
?rotg	s, d, c, z	Generate Givens rotation of points (routines)
?rotm	s, d	Modified Givens plane rotation of points (routines)
?rotmg	s, d	Generate modified Givens plane rotation of points (routines)
?scal	s, d, c, z, cs, zd	Vector-scalar product (routines)
?swap	s, d, c, z	Vector-vector swap (routines)
i?amax	s, d, c, z	Index of the maximum absolute value element of a vector (functions)
i?amin	s, d, c, z	Index of the minimum absolute value element of a vector (functions)

Routine or Function Group	Data Types	Description
?cabs1	s, d	Auxiliary functions, compute the absolute value of a complex number of single or double precision

?asum

Computes the sum of magnitudes of the vector elements.

Syntax

FORTRAN 77:

```
res = sasum(n, x, incx)
res = scasum(n, x, incx)
res = dasum(n, x, incx)
res = dzasum(n, x, incx)
```

Fortran 95:

```
res = asum(x)
```

C:

```
float cblas_sasum (const MKL_INT n, const float *x, const MKL_INT incx);
float cblas_scasum (const MKL_INT n, const void *x, const MKL_INT incx);
double cblas_dasum (const MKL_INT n, const double *x, const MKL_INT incx);
double cblas_dzasum (const MKL_INT n, const void *x, const MKL_INT incx);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?asum routine computes the sum of the magnitudes of elements of a real vector, or the sum of magnitudes of the real and imaginary parts of elements of a complex vector:

```
res = |Re x1| + |Im x1| + |Re x2| + |Im x2| + ... + |Re xn| + |Im xn|,
```

where x is a vector with n elements.

Input Parameters

n	INTEGER. Specifies the number of elements in vector x.
x	REAL for sasum DOUBLE PRECISION for dasum COMPLEX for scasum DOUBLE COMPLEX for dzasum

Array, size at least $(1 + (n-1) * \text{abs}(incx))$.

incx INTEGER. Specifies the increment for indexing vector *x*.

Output Parameters

<i>res</i>	REAL for sasum
	DOUBLE PRECISION for dasum
	REAL for scasum
	DOUBLE PRECISION for dzasum
	Contains the sum of magnitudes of real and imaginary parts of all elements of the vector.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `asum` interface are the following:

<i>x</i>	Holds the array of size <i>n</i> .
----------	------------------------------------

?axpy

Computes a vector-scalar product and adds the result to a vector.

Syntax

FORTRAN 77:

```
call saxpy(n, a, x, incx, y, incy)
call daxpy(n, a, x, incx, y, incy)
call caxpy(n, a, x, incx, y, incy)
call zaxpy(n, a, x, incx, y, incy)
```

Fortran 95:

```
call axpy(x, y [, a])
```

C:

```
void cblas_saxpy (const MKL_INT n, const float a, const float *x, const MKL_INT incx,
float *y, const MKL_INT incy);
void cblas_daxpy (const MKL_INT n, const double a, const double *x, const MKL_INT incx,
double *y, const MKL_INT incy);
void cblas_caxpy (const MKL_INT n, const void *a, const void *x, const MKL_INT incx,
void *y, const MKL_INT incy);
void cblas_zaxpy (const MKL_INT n, const void *a, const void *x, const MKL_INT incx,
void *y, const MKL_INT incy);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?axpy routines perform a vector-vector operation defined as

```
y := a*x + y
```

where:

a is a scalar

x and *y* are vectors each with a number of elements that equals *n*.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vectors <i>x</i> and <i>y</i> .
<i>a</i>	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Specifies the scalar <i>a</i> .
<i>x</i>	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Array, size at least (1 + (n-1)*abs(incx)).
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .
<i>y</i>	REAL for saxpy DOUBLE PRECISION for daxpy COMPLEX for caxpy DOUBLE COMPLEX for zaxpy Array, size at least (1 + (n-1)*abs(incy)).
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .

Output Parameters

<i>y</i>	Contains the updated vector <i>y</i> .
----------	--

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `axpy` interface are the following:

<code>x</code>	Holds the array of size n .
<code>y</code>	Holds the array of size n .
<code>a</code>	The default value is 1.

?copy

Copies vector to another vector.

Syntax

FORTRAN 77:

```
call scopy(n, x, incx, y, incy)
call dcopy(n, x, incx, y, incy)
call ccopy(n, x, incx, y, incy)
call zcopy(n, x, incx, y, incy)
```

Fortran 95:

```
call copy(x, y)
```

C:

```
void cblas_scopy (const MKL_INT n, const float *x, const MKL_INT incx, float *y, const
MKL_INT incy);
void cblas_dcopy (const MKL_INT n, const double *x, const MKL_INT incx, double *y,
const MKL_INT incy);
void cblas_ccopy (const MKL_INT n, const void *x, const MKL_INT incx, void *y, const
MKL_INT incy);
void cblas_zcopy (const MKL_INT n, const void *x, const MKL_INT incx, void *y, const
MKL_INT incy);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `blas.f90`
- C: `mkl.h`

Description

The `?copy` routines perform a vector-vector operation defined as

```
y = x,
```

where x and y are vectors.

Input Parameters

<code>n</code>	INTEGER. Specifies the number of elements in vectors x and y .
<code>x</code>	REAL for <code>scopy</code> DOUBLE PRECISION for <code>dcopy</code>

COMPLEX for `ccopy`
 DOUBLE COMPLEX for `zcopy`
 Array, size at least $(1 + (n-1) * \text{abs}(incx))$.

incx INTEGER. Specifies the increment for the elements of *x*.

y REAL for `scopy`
 DOUBLE PRECISION for `dcopy`
 COMPLEX for `ccopy`
 DOUBLE COMPLEX for `zcopy`
 Array, size at least $(1 + (n-1) * \text{abs}(incy))$.

incy INTEGER. Specifies the increment for the elements of *y*.

Output Parameters

y Contains a copy of the vector *x* if *n* is positive. Otherwise, parameters are unaltered.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `copy` interface are the following:

x Holds the vector with the number of elements *n*.
y Holds the vector with the number of elements *n*.

?dot

Computes a vector-vector dot product.

Syntax

FORTRAN 77:

```
res = sdot(n, x, incx, y, incy)
res = ddot(n, x, incx, y, incy)
```

Fortran 95:

```
res = dot(x, y)
```

C:

```
float cblas_sdot (const MKL_INT n, const float *x, const MKL_INT incx, const float *y,
const MKL_INT incy);

double cblas_ddot (const MKL_INT n, const double *x, const MKL_INT incx, const double
*y, const MKL_INT incy);
```

Include Files

- Fortran: `mkl.fi`

- Fortran 95: blas.f90
- C: mkl.h

Description

The ?dot routines perform a vector-vector reduction operation defined as

$$res = \sum_{i=1}^n x_i * y_i,$$

where x_i and y_i are elements of vectors x and y .

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vectors x and y .
<i>x</i>	REAL for sdot DOUBLE PRECISION for ddot Array, size at least $(1+(n-1)*abs(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of x .
<i>y</i>	REAL for sdot DOUBLE PRECISION for ddot Array, size at least $(1+(n-1)*abs(incy))$.
<i>incy</i>	INTEGER. Specifies the increment for the elements of y .

Output Parameters

<i>res</i>	REAL for sdot DOUBLE PRECISION for ddot Contains the result of the dot product of x and y , if n is positive. Otherwise, <i>res</i> contains 0.
------------	---

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `dot` interface are the following:

<i>x</i>	Holds the vector with the number of elements n .
<i>y</i>	Holds the vector with the number of elements n .

?sdot

Computes a vector-vector dot product with double precision.

Syntax

FORTRAN 77:

```
res = sdsdot(n, sb, sx, incx, sy, incy)
res = dsdot(n, sx, incx, sy, incy)
```

Fortran 95:

```
res = sdot(sx, sy)
res = sdot(sx, sy, sb)
```

C:

```
float cblas_sdsdot (const MKL_INT n, const float sb, const float *sx, const MKL_INT
incx, const float *sy, const MKL_INT incy);

double cblas_dsdot (const MKL_INT n, const float *sx, const MKL_INT incx, const float
*sy, const MKL_INT incy);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?sdot routines compute the inner product of two vectors with double precision. Both routines use double precision accumulation of the intermediate results, but the `sdsdot` routine outputs the final result in single precision, whereas the `dsdot` routine outputs the double precision result. The function `sdsdot` also adds scalar value `sb` to the inner product.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in the input vectors <code>sx</code> and <code>sy</code> .
<i>sb</i>	REAL. Single precision scalar to be added to inner product (for the function <code>sdsdot</code> only).
<i>sx</i> , <i>sy</i>	REAL. Arrays, size at least $(1+(n-1)*\text{abs}(incx))$ and $(1+(n-1)*\text{abs}(incy))$, respectively. Contain the input single precision vectors.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <code>sx</code> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <code>sy</code> .

Output Parameters

<i>res</i>	REAL for <code>sdsdot</code> DOUBLE PRECISION for <code>dsdot</code> Contains the result of the dot product of <code>sx</code> and <code>sy</code> (with <code>sb</code> added for <code>sdsdot</code>), if <i>n</i> is positive. Otherwise, <i>res</i> contains <code>sb</code> for <code>sdsdot</code> and 0 for <code>dsdot</code> .
------------	--

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sdot` interface are the following:

<code>sx</code>	Holds the vector with the number of elements n .
<code>sy</code>	Holds the vector with the number of elements n .

NOTE

Note that scalar parameter `sb` is declared as a required parameter in Fortran 95 interface for the function `sdot` to distinguish between function flavors that output final result in different precision.

?dotc

Computes a dot product of a conjugated vector with another vector.

Syntax

FORTRAN 77:

```
res = cdotc(n, x, incx, y, incy)
res = zdotc(n, x, incx, y, incy)
```

Fortran 95:

```
res = dotc(x, y)
```

C:

```
void cblas_cdotc_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void
*y, const MKL_INT incy, void *dotc);
void cblas_zdotc_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void
*y, const MKL_INT incy, void *dotc);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `blas.f90`
- C: `mkl.h`

Description

The `?dotc` routines perform a vector-vector operation defined as:

$$res = \sum_{i=1}^n \text{conjg}(x_i) * y_i,$$

where x_i and y_i are elements of vectors x and y .

Input Parameters

<code>n</code>	INTEGER. Specifies the number of elements in vectors x and y .
<code>x</code>	COMPLEX for <code>cdotc</code> DOUBLE COMPLEX for <code>zdotc</code>
	Array, size at least $(1 + (n - 1) * \text{abs}(incx))$.

<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .
<i>y</i>	COMPLEX for cdotc DOUBLE COMPLEX for zdotc Array, size at least $(1 + (n - 1) * \text{abs}(incy))$.
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .

Output Parameters

<i>res</i>	COMPLEX for cdotc DOUBLE COMPLEX for zdotc Contains the result of the dot product of the conjugated <i>x</i> and unconjugated <i>y</i> , if <i>n</i> is positive. Otherwise, it contains 0.
------------	---

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `dotc` interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .

?dotu

Computes a vector-vector dot product.

Syntax

FORTRAN 77:

```
res = cdotu(n, x, incx, y, incy)
res = zdotu(n, x, incx, y, incy)
```

Fortran 95:

```
res = dotu(x, y)
```

C:

```
void cblas_cdotu_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *dotu);
void cblas_zdotu_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *dotu);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The `?dotu` routines perform a vector-vector reduction operation defined as

$$res = \sum_{i=1}^n x_i * y_i$$

where x_i and y_i are elements of complex vectors x and y .

Input Parameters

<code>n</code>	INTEGER. Specifies the number of elements in vectors x and y .
<code>x</code>	COMPLEX for <code>cdotu</code> DOUBLE COMPLEX for <code>zdotu</code> Array, size at least $(1 + (n - 1) * \text{abs}(incx))$.
<code>incx</code>	INTEGER. Specifies the increment for the elements of x .
<code>y</code>	COMPLEX for <code>cdotu</code> DOUBLE COMPLEX for <code>zdotu</code> Array, size at least $(1 + (n - 1) * \text{abs}(incy))$.
<code>incy</code>	INTEGER. Specifies the increment for the elements of y .

Output Parameters

<code>res</code>	COMPLEX for <code>cdotu</code> DOUBLE COMPLEX for <code>zdotu</code> Contains the result of the dot product of x and y , if n is positive. Otherwise, it contains 0.
------------------	--

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `dotu` interface are the following:

<code>x</code>	Holds the vector with the number of elements n .
<code>y</code>	Holds the vector with the number of elements n .

?nrm2

Computes the Euclidean norm of a vector.

Syntax

FORTRAN 77:

```
res = snrm2(n, x, incx)
res = dnrm2(n, x, incx)
res = scnrm2(n, x, incx)
```

```
res = dznrm2(n, x, incx)
```

Fortran 95:

```
res = nrm2(x)
```

C:

```
float cblas_snrm2 (const MKL_INT n, const float *x, const MKL_INT incx);
double cblas_dnrm2 (const MKL_INT n, const double *x, const MKL_INT incx);
float cblas_scnrm2 (const MKL_INT n, const void *x, const MKL_INT incx);
double cblas_dznrm2 (const MKL_INT n, const void *x, const MKL_INT incx);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?nrm2 routines perform a vector reduction operation defined as

```
res = ||x||,
```

where:

x is a vector,

res is a value containing the Euclidean norm of the elements of *x*.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vector <i>x</i> .
<i>x</i>	REAL for snrm2 DOUBLE PRECISION for dnrm2 COMPLEX for scnrm2 DOUBLE COMPLEX for dznrm2 Array, size at least (1 + (n -1)*abs (incx)).
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

Output Parameters

<i>res</i>	REAL for snrm2 DOUBLE PRECISION for dnrm2 REAL for scnrm2 DOUBLE PRECISION for dznrm2 Contains the Euclidean norm of the vector <i>x</i> .
------------	--

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `nrm2` interface are the following:

`x` Holds the vector with the number of elements `n`.

?rot

Performs rotation of points in the plane.

Syntax

FORTRAN 77:

```
call srot(n, x, incx, y, incy, c, s)
call drot(n, x, incx, y, incy, c, s)
call csrot(n, x, incx, y, incy, c, s)
call zdrot(n, x, incx, y, incy, c, s)
```

Fortran 95:

```
call rot(x, y, c, s)
```

C:

```
void cblas_srot (const MKL_INT n, float *x, const MKL_INT incx, float *y, const MKL_INT incy, const float c, const float s);
void cblas_drot (const MKL_INT n, double *x, const MKL_INT incx, double *y, const MKL_INT incy, const double c, const double s);
void cblas_csrot (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT incy, const float c, const float s);
void cblas_zdrot (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT incy, const double c, const double s);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `blas.f90`
- C: `mkl.h`

Description

Given two complex vectors x and y , each vector element of these vectors is replaced as follows:

$$x_i = c^*x_i + s^*y_i$$

$$y_i = c^*y_i - s^*x_i$$

Input Parameters

`n` INTEGER. Specifies the number of elements in vectors x and y .

`x` REAL for `srot`

DOUBLE PRECISION for `drot`

COMPLEX for `csrot`

DOUBLE COMPLEX for `zdrot`

Array, size at least $(1 + (n-1) * \text{abs}(incx))$.

<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .
<i>y</i>	REAL for srot DOUBLE PRECISION for drot COMPLEX for csrot DOUBLE COMPLEX for zdrot Array, size at least $(1 + (n - 1) * \text{abs}(incy))$.
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .
<i>c</i>	REAL for srot DOUBLE PRECISION for drot REAL for csrot DOUBLE PRECISION for zdrot A scalar.
<i>s</i>	REAL for srot DOUBLE PRECISION for drot REAL for csrot DOUBLE PRECISION for zdrot A scalar.

Output Parameters

<i>x</i>	Each element is replaced by $c*x + s*y$.
<i>y</i>	Each element is replaced by $c*y - s*x$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `rot` interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .

?rotg

Computes the parameters for a Givens rotation.

Syntax

FORTRAN 77:

```
call srotg(a, b, c, s)
call drotg(a, b, c, s)
call crotg(a, b, c, s)
```

```
call zrotg(a, b, c, s)
```

Fortran 95:

```
call rotg(a, b, c, s)
```

C:

```
void cblas_srotg (float *a, float *b, float *c, float *s);
void cblas_drotg (double *a, double *b, double *c, double *s);
void cblas_crotg (void *a, const void *b, float *c, void *s);
void cblas_zrotg (void *a, const void *b, double *c, void *s);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

Given the Cartesian coordinates (a, b) of a point, these routines return the parameters c, s, r , and z associated with the Givens rotation. The parameters c and s define a unitary matrix such that:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

The parameter z is defined such that if $|a| > |b|$, z is s ; otherwise if c is not 0 z is $1/c$; otherwise z is 1.

See a more accurate LAPACK version [?lartg](#).

Input Parameters

a REAL **for** srotg
 DOUBLE PRECISION **for** drotg
 COMPLEX **for** crotg
 DOUBLE COMPLEX **for** zrotg
 Provides the x-coordinate of the point p.

b REAL **for** srotg
 DOUBLE PRECISION **for** drotg
 COMPLEX **for** crotg
 DOUBLE COMPLEX **for** zrotg
 Provides the y-coordinate of the point p.

Output Parameters

a Contains the parameter r associated with the Givens rotation.
b Contains the parameter z associated with the Givens rotation.
c REAL **for** srotg

```

DOUBLE PRECISION for drotg
REAL for crotg
DOUBLE PRECISION for zrotg
Contains the parameter c associated with the Givens rotation.

s
REAL for srotg
DOUBLE PRECISION for drotg
COMPLEX for crotg
DOUBLE COMPLEX for zrotg
Contains the parameter s associated with the Givens rotation.
```

?rotm

Performs modified Givens rotation of points in the plane.

Syntax**FORTRAN 77:**

```
call srotm(n, x, incx, y, incy, param)
call drotm(n, x, incx, y, incy, param)
```

Fortran 95:

```
call rotm(x, y, param)
```

C:

```
void cblas_srotm (const MKL_INT n, float *x, const MKL_INT incx, float *y, const
MKL_INT incy, const float *param);
void cblas_drotm (const MKL_INT n, double *x, const MKL_INT incx, double *y, const
MKL_INT incy, const double *param);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

Given two vectors *x* and *y*, each vector element of these vectors is replaced as follows:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = H \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

for *i*=1 to *n*, where *H* is a modified Givens transformation matrix whose values are stored in the *param*(2) through *param*(5) array. See discussion on the *param* argument.

Input Parameters

n INTEGER. Specifies the number of elements in vectors *x* and *y*.

x REAL for srotm

DOUBLE PRECISION for drotm

Array, size at least $(1 + (n - 1) * \text{abs}(incx))$.

incx INTEGER. Specifies the increment for the elements of *x*.

y REAL for srotm

DOUBLE PRECISION for drotm

Array, size at least $(1 + (n - 1) * \text{abs}(incy))$.

incy INTEGER. Specifies the increment for the elements of *y*.

param REAL for srotm

DOUBLE PRECISION for drotm

Array, size 5.

The elements of the *param* array are:

param(1) contains a switch, *flag*. *param*(2-5) contain *h11*, *h21*, *h12*, and *h22*, respectively, the components of the array *H*.

Depending on the values of *flag*, the components of *H* are set as follows:

$$\text{flag} = -1. : H = \begin{bmatrix} h11 & h12 \\ h21 & h22 \end{bmatrix}$$

...i!!!!!! ...i!!! ...i!!!!!!

$$\text{flag} = 1. : H = \begin{bmatrix} h11 & 1. \\ -1 & h22 \end{bmatrix}$$

...i!!!!!! ...i!!! ...i!!!!!!

In the last three cases, the matrix entries of 1., -1., and 0. are assumed based on the value of *flag* and are not required to be set in the *param* vector.

Output Parameters

x Each element $x(i)$ is replaced by $h11*x(i) + h12*y(i)$.

y Each element $y(i)$ is replaced by $h21*x(i) + h22*y(i)$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `rotm` interface are the following:

x Holds the vector with the number of elements *n*.

y Holds the vector with the number of elements *n*.

?rotmg

Computes the parameters for a modified Givens rotation.

Syntax

FORTRAN 77:

```
call srotmg(d1, d2, x1, y1, param)
call drotmg(d1, d2, x1, y1, param)
```

Fortran 95:

```
call rotmg(d1, d2, x1, y1, param)
```

C:

```
void cblas_srotmg (float *d1, float *d2, float *x1, const float y1, float *param);
void cblas_drotmg (double *d1, double *d2, double *x1, const double y1, double *param);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `blas.f90`
- C: `mkl.h`

Description

Given Cartesian coordinates $(x1, y1)$ of an input vector, these routines compute the components of a modified Givens transformation matrix H that zeros the y -component of the resulting vector:

$$\begin{bmatrix} x1 \\ 0 \end{bmatrix} = H \begin{bmatrix} x1\sqrt{d1} \\ y1\sqrt{d1} \end{bmatrix}$$

Input Parameters

<i>d1</i>	REAL for srotmg DOUBLE PRECISION for drotmg Provides the scaling factor for the <i>x</i> -coordinate of the input vector.
<i>d2</i>	REAL for srotmg DOUBLE PRECISION for drotmg Provides the scaling factor for the <i>y</i> -coordinate of the input vector.
<i>x1</i>	REAL for srotmg DOUBLE PRECISION for drotmg Provides the <i>x</i> -coordinate of the input vector.
<i>y1</i>	REAL for srotmg DOUBLE PRECISION for drotmg Provides the <i>y</i> -coordinate of the input vector.

Output Parameters

<i>d1</i>	REAL for srotmg DOUBLE PRECISION for drotmg Provides the first diagonal element of the updated matrix.
<i>d2</i>	REAL for srotmg DOUBLE PRECISION for drotmg Provides the second diagonal element of the updated matrix.
<i>x1</i>	REAL for srotmg DOUBLE PRECISION for drotmg Provides the <i>x</i> -coordinate of the rotated vector before scaling.
<i>param</i>	REAL for srotmg DOUBLE PRECISION for drotmg Array, size 5. The elements of the <i>param</i> array are: <i>param</i> (1) contains a switch, <i>flag</i> . <i>param</i> (2–5) contain <i>h11</i> , <i>h21</i> , <i>h12</i> , and <i>h22</i> , respectively, the components of the array <i>H</i> . Depending on the values of <i>flag</i> , the components of <i>H</i> are set as follows:

$$\text{flag} = -1. : H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

[View Source](#) | [Download ZIP](#)

$$flag = 1. : H = \begin{bmatrix} h_{11} & 1. \\ -1 & h_{22} \end{bmatrix}$$

[View Source](#) | [Download ZIP](#)

In the last three cases, the matrix entries of 1., -1., and 0. are assumed based on the value of *flag* and are not required to be set in the *param* vector.

?scal

Computes the product of a vector by a scalar.

Syntax

FORTRAN 77:

```
call sscal(n, a, x, incx)
call dscal(n, a, x, incx)
call cscal(n, a, x, incx)
call zscal(n, a, x, incx)
call csscal(n, a, x, incx)
call zdscal(n, a, x, incx)
```

Fortran 95:

```
call scal(x, a)
```

C:

```
void cblas_sscal (const MKL_INT n, const float a, float *x, const MKL_INT incx);
void cblas_dscal (const MKL_INT n, const double a, double *x, const MKL_INT incx);
void cblas_cscal (const MKL_INT n, const void *a, void *x, const MKL_INT incx);
void cblas_zscal (const MKL_INT n, const void *a, void *x, const MKL_INT incx);
void cblas_csscal (const MKL_INT n, const float a, void *x, const MKL_INT incx);
void cblas_zdscal (const MKL_INT n, const double a, void *x, const MKL_INT incx);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?scal routines perform a vector operation defined as

```
x = a*x
```

where:

a is a scalar, *x* is an *n*-element vector.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vector <i>x</i> .
<i>a</i>	REAL for sscal and csscal DOUBLE PRECISION for dscal and zdscal COMPLEX for cscal DOUBLE COMPLEX for zscal Specifies the scalar <i>a</i> .
<i>x</i>	REAL for sscal DOUBLE PRECISION for dscal COMPLEX for cscal and csscal DOUBLE COMPLEX for zscal and zdscal Array, size at least (1 + (n -1)*abs(<i>incx</i>)).
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

Output Parameters

<i>x</i>	Updated vector <i>x</i> .
----------	---------------------------

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine scal interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>n</i> .
----------	---

?swap

Swaps a vector with another vector.

Syntax

FORTRAN 77:

```
call sswap(n, x, incx, y, incy)
call dswap(n, x, incx, y, incy)
call cswap(n, x, incx, y, incy)
call zswap(n, x, incx, y, incy)
```

Fortran 95:

```
call swap(x, y)
```

C:

```
void cblas_sswap (const MKL_INT n, float *x, const MKL_INT incx, float *y, const
MKL_INT incy);

void cblas_dswap (const MKL_INT n, double *x, const MKL_INT incx, double *y, const
MKL_INT incy);

void cblas_cswap (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT
incy);

void cblas_zswap (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT
incy);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

Given two vectors x and y , the ?swap routines return vectors y and x swapped, each replacing the other.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vectors x and y .
<i>x</i>	REAL for sswap DOUBLE PRECISION for dswap COMPLEX for cswap DOUBLE COMPLEX for zswap Array, size at least $(1 + (n-1) * \text{abs}(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of x .
<i>y</i>	REAL for sswap DOUBLE PRECISION for dswap COMPLEX for cswap DOUBLE COMPLEX for zswap Array, size at least $(1 + (n-1) * \text{abs}(incy))$.
<i>incy</i>	INTEGER. Specifies the increment for the elements of y .

Output Parameters

<i>x</i>	Contains the resultant vector x , that is, the input vector y .
<i>y</i>	Contains the resultant vector y , that is, the input vector x .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `swap` interface are the following:

<code>x</code>	Holds the vector with the number of elements n .
<code>y</code>	Holds the vector with the number of elements n .

i?amax

Finds the index of the element with maximum absolute value.

Syntax

FORTRAN 77:

```
index = isamax(n, x, incx)
index = idamax(n, x, incx)
index = icamax(n, x, incx)
index = izamax(n, x, incx)
```

Fortran 95:

```
index = iamax(x)
```

C:

```
CBLAS_INDEX cblas_isamax (const MKL_INT n, const float *x, const MKL_INT incx);
CBLAS_INDEX cblas_idamax (const MKL_INT n, const double *x, const MKL_INT incx);
CBLAS_INDEX cblas_icamax (const MKL_INT n, const void *x, const MKL_INT incx);
CBLAS_INDEX cblas_izamax (const MKL_INT n, const void *x, const MKL_INT incx);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `blas.f90`
- C: `mkl.h`

Description

Given a vector x , the `i?amax` functions return the position of the vector element $x(i)$ that has the largest absolute value for real flavors, or the largest sum $|\text{Re}(x(i))| + |\text{Im}(x(i))|$ for complex flavors.

If n is not positive, 0 is returned.

If more than one vector element is found with the same largest absolute value, the index of the first one encountered is returned.

Input Parameters

<code>n</code>	INTEGER. Specifies the number of elements in vector x .
<code>x</code>	REAL for <code>isamax</code> DOUBLE PRECISION for <code>idamax</code> COMPLEX for <code>icamax</code> DOUBLE COMPLEX for <code>izamax</code> Array, size at least $(1+(n-1) * \text{abs}(incx))$.

incx INTEGER. Specifies the increment for the elements of *x*.

Output Parameters

index INTEGER. Contains the position of vector element that has the largest absolute value such that *x(index)* has the largest absolute value.

Fortran 95 Interface Notes

Functions and routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the function `iamax` interface are the following:

x Holds the vector with the number of elements *n*.

i?amin

Finds the index of the element with the smallest absolute value.

Syntax

FORTRAN 77:

```
index = isamin(n, x, incx)
index = idamin(n, x, incx)
index = icamin(n, x, incx)
index = izamin(n, x, incx)
```

Fortran 95:

```
index = iamin(x)
```

C:

```
CBLAS_INDEX cblas_isamin (const MKL_INT n, const float *x, const MKL_INT incx);
CBLAS_INDEX cblas_idamin (const MKL_INT n, const double *x, const MKL_INT incx);
CBLAS_INDEX cblas_icamin (const MKL_INT n, const void *x, const MKL_INT incx);
CBLAS_INDEX cblas_izamin (const MKL_INT n, const void *x, const MKL_INT incx);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `blas.f90`
- C: `mkl.h`

Description

Given a vector *x*, the `i?amin` functions return the position of the vector element *x(i)* that has the smallest absolute value for real flavors, or the smallest sum $|\operatorname{Re}(x(i))| + |\operatorname{Im}(x(i))|$ for complex flavors.

If *n* is not positive, 0 is returned.

If more than one vector element is found with the same smallest absolute value, the index of the first one encountered is returned.

Input Parameters

<i>n</i>	INTEGER. On entry, <i>n</i> specifies the number of elements in vector <i>x</i> .
<i>x</i>	REAL for <i>isamin</i> DOUBLE PRECISION for <i>idamin</i> COMPLEX for <i>icamin</i> DOUBLE COMPLEX for <i>izamin</i> Array, size at least $(1 + (n-1) * \text{abs}(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

Output Parameters

<i>index</i>	INTEGER. Indicates the position of vector element with the smallest absolute value such that <i>x(index)</i> has the smallest absolute value.
--------------	---

Fortran 95 Interface Notes

Functions and routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the function *iamin* interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>n</i> .
----------	---

?cabs1

Computes absolute value of complex number.

Syntax

FORTRAN 77:

```
res = scabs1(z)
res = dcabs1(z)
```

Fortran 95:

```
res = cabs1(z)
```

C:

```
float cblas_scabs1 (const void *z);
double cblas_dcabs1 (const void *z);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The `?cabs1` is an auxiliary routine for a few BLAS Level 1 routines. This routine performs an operation defined as

```
res=|Re(z)|+|Im(z)|,
```

where z is a scalar, and res is a value containing the absolute value of a complex number z .

Input Parameters

z	COMPLEX scalar for <code>scabs1</code> .
	DOUBLE COMPLEX scalar for <code>dcabs1</code> .

Output Parameters

res	REAL for <code>scabs1</code> .
	DOUBLE PRECISION for <code>dcabs1</code> .
	Contains the absolute value of a complex number z .

BLAS Level 2 Routines

This section describes BLAS Level 2 routines, which perform matrix-vector operations. [Table “BLAS Level 2 Routine Groups and Their Data Types”](#) lists the BLAS Level 2 routine groups and the data types associated with them.

BLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
<code>?gbmv</code>	s, d, c, z	Matrix-vector product using a general band matrix
<code>?gemv</code>	s, d, c, z	Matrix-vector product using a general matrix
<code>?ger</code>	s, d	Rank-1 update of a general matrix
<code>?gerc</code>	c, z	Rank-1 update of a conjugated general matrix
<code>?geru</code>	c, z	Rank-1 update of a general matrix, unconjugated
<code>?hbmv</code>	c, z	Matrix-vector product using a Hermitian band matrix
<code>?hemv</code>	c, z	Matrix-vector product using a Hermitian matrix
<code>?her</code>	c, z	Rank-1 update of a Hermitian matrix
<code>?her2</code>	c, z	Rank-2 update of a Hermitian matrix
<code>?hpmv</code>	c, z	Matrix-vector product using a Hermitian packed matrix
<code>?hpr</code>	c, z	Rank-1 update of a Hermitian packed matrix
<code>?hpr2</code>	c, z	Rank-2 update of a Hermitian packed matrix
<code>?sbmv</code>	s, d	Matrix-vector product using symmetric band matrix
<code>?spmv</code>	s, d	Matrix-vector product using a symmetric packed matrix
<code>?spr</code>	s, d	Rank-1 update of a symmetric packed matrix
<code>?spr2</code>	s, d	Rank-2 update of a symmetric packed matrix
<code>?symv</code>	s, d	Matrix-vector product using a symmetric matrix

Routine Groups	Data Types	Description
?syr	s, d	Rank-1 update of a symmetric matrix
?syr2	s, d	Rank-2 update of a symmetric matrix
?tbtmv	s, d, c, z	Matrix-vector product using a triangular band matrix
?tbsv	s, d, c, z	Solution of a linear system of equations with a triangular band matrix
?tpmv	s, d, c, z	Matrix-vector product using a triangular packed matrix
?tpsv	s, d, c, z	Solution of a linear system of equations with a triangular packed matrix
?trmv	s, d, c, z	Matrix-vector product using a triangular matrix
?trs	s, d, c, z	Solution of a linear system of equations with a triangular matrix

?gbmv

Computes a matrix-vector product using a general band matrix

Syntax**FORTRAN 77:**

```
call sgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
call dgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
call cgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
call zgbmv(trans, m, n, kl, ku, alpha, a, lda, x, incx, beta, y, incy)
```

Fortran 95:

```
call gbm(a, x, y [,kl] [,m] [,alpha] [,beta] [,trans])
```

C:

```
void cblas_sgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const float alpha, const float *a, const MKL_INT lda, const float *x, const MKL_INT incx, const float beta, float *y, const MKL_INT incy);

void cblas_dgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const double alpha, const double *a, const MKL_INT lda, const double *x, const MKL_INT incx, const double beta, double *y, const MKL_INT incy);

void cblas_cgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);

void cblas_zgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?gbmv routines perform a matrix-vector operation defined as

$$y := \alpha * A * x + \beta * y,$$

or

$$y := \alpha * A' * x + \beta * y,$$

or

$$y := \alpha * \text{conjg}(A') * x + \beta * y,$$

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*n* band matrix, with *kl* sub-diagonals and *ku* super-diagonals.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>trans</i>	CHARACTER*1. Specifies the operation: If <i>trans</i> = 'N' or 'n', then $y := \alpha * A * x + \beta * y$ If <i>trans</i> = 'T' or 't', then $y := \alpha * A' * x + \beta * y$ If <i>trans</i> = 'C' or 'c', then $y := \alpha * \text{conjg}(A') * x + \beta * y$
<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>kl</i>	INTEGER. Specifies the number of sub-diagonals of the matrix <i>A</i> . The value of <i>kl</i> must satisfy $0 \leq kl$.
<i>ku</i>	INTEGER. Specifies the number of super-diagonals of the matrix <i>A</i> . The value of <i>ku</i> must satisfy $0 \leq ku$.
<i>alpha</i>	REAL for sgbmv DOUBLE PRECISION for dgbmv COMPLEX for cgbmv DOUBLE COMPLEX for zgbmv Specifies the scalar <i>alpha</i> .

a

```
REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv
Array, size (lda, n).
```

Before entry, the leading $(k_l + k_u + 1)$ by *n* part of the array *a* must contain the matrix of coefficients. This matrix must be supplied column-by-column, with the leading diagonal of the matrix in row $(k_u + 1)$ of the array, the first super-diagonal starting at position 2 in row *k_u*, the first sub-diagonal starting at position 1 in row $(k_u + 2)$, and so on. Elements in the array *a* that do not correspond to elements in the band matrix (such as the top left *k_u* by *k_u* triangle) are not referenced.

The following program segment transfers a band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
do 20, j = 1, n
      k = k_u + 1 - j
      do 10, i = max(1, j-k_u), min(m, j+k_l)
            a(k+i, j) = matrix(i,j)
10          continue
20          continue
```

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least $(k_l + k_u + 1)$.

x

```
REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv
Array, size at least  $(1 + (n - 1) * \text{abs}(incx))$  when trans= 'N' or 'n', and at least  $(1 + (m - 1) * \text{abs}(incx))$  otherwise. Before entry, the array x must contain the vector x.
```

incx INTEGER. Specifies the increment for the elements of *x*. *incx* must not be zero.

beta

```
REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv
Specifies the scalar beta. When beta is equal to zero, then y need not be set on input.
```

y

```
REAL for sgbmv
DOUBLE PRECISION for dgbmv
COMPLEX for cgbmv
DOUBLE COMPLEX for zgbmv
```

Array, size at least $(1 + (m - 1) * \text{abs}(incy))$ when $\text{trans} = 'N'$ or ' n ' and at least $(1 + (n - 1) * \text{abs}(incy))$ otherwise. Before entry, the incremented array y must contain the vector y .

incy INTEGER. Specifies the increment for the elements of y .
The value of *incy* must not be zero.

Output Parameters

y Updated vector y .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbmv` interface are the following:

<i>a</i>	Holds the array <i>a</i> of size $(kl+ku+1, n)$. Contains a banded matrix $m*n$ with kl lower diagonal and ku upper diagonal.
<i>x</i>	Holds the vector with the number of elements <i>rx</i> , where $rx = n$ if $\text{trans} = 'N'$, $rx = m$ otherwise.
<i>y</i>	Holds the vector with the number of elements <i>ry</i> , where $ry = m$ if $\text{trans} = 'N'$, $ry = n$ otherwise.
<i>trans</i>	Must be ' N ', ' C ', or ' T '. The default value is ' N '.
<i>kl</i>	If omitted, assumed $kl = ku$, that is, the number of lower diagonals equals the number of the upper diagonals.
<i>ku</i>	Restored as $ku = lda - kl - 1$, where <i>lda</i> is the leading dimension of matrix <i>A</i> .
<i>m</i>	If omitted, assumed $m = n$, that is, a square matrix.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?gemv

Computes a matrix-vector product using a general matrix

Syntax

FORTRAN 77:

```
call sgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call dgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call cgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call zgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

```
call scgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call dzgemv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

Fortran 95:

```
call gemv(a, x, y [,alpha] [,beta] [,trans])
```

C:

```

void cblas_sgmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const float alpha, const float *a, const MKL_INT lda, const float *x, const MKL_INT incx, const float beta, float *y, const MKL_INT incy);

void cblas_dgemv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const double alpha, const double *a, const MKL_INT lda, const double *x, const MKL_INT incx, const double beta, double *y, const MKL_INT incy);

void cblas_cgmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);

void cblas_zgmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);

```

Include Files

- Fortran: mkl.fi
 - Fortran 95: blas.f90
 - C: mkl.h

Description

The `?gemv` routines perform a matrix-vector operation defined as

$y := alpha * A * x + beta * y,$

or

$y := alpha * A' * x + beta * y,$

or

$y := \text{alpha}^* \text{conjg}(A')^* x + \text{beta}^* y,$

where:

alpha and *beta* are scalars,

x and y are vectors,

A is an m -by- n matrix.

Input Parameters

Layout

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

trans

CHARACTER*1. Specifies the operation:

if $trans = 'N'$ or $'n'$, then $y := alpha * A * x + beta * y$;

if $trans = 'T'$ or $'t'$, then $y := alpha * A' * x + beta * y$

if *trans*= 'C' or 'c', then *y* := *alpha* *conjg(*A*')**x* + *beta***y*.

m INTEGER. Specifies the number of rows of the matrix *A*. The value of *m* must be at least zero.

n INTEGER. Specifies the number of columns of the matrix *A*. The value of *n* must be at least zero.

alpha REAL **for** sgemv

DOUBLE PRECISION **for** dgemv

COMPLEX **for** cgemv, scgemv

DOUBLE COMPLEX **for** zgemv, dzgemv

Specifies the scalar *alpha*.

a REAL **for** sgemv, scgemv

DOUBLE PRECISION **for** dgemv, dzgemv

COMPLEX **for** cgemv

DOUBLE COMPLEX **for** zgemv

Array, size (*lda*, *n*). Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients.

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least max(1, *m*).

x REAL **for** sgemv

DOUBLE PRECISION **for** dgemv

COMPLEX **for** cgemv, scgemv

DOUBLE COMPLEX **for** zgemv, dzgemv

Array, size at least (1+(*n*-1)*abs(*incx*)) when *trans*= 'N' or 'n' and at least (1+(*m* - 1)*abs(*incx*)) otherwise. Before entry, the incremented array *x* must contain the vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

beta REAL **for** sgemv

DOUBLE PRECISION **for** dgemv

COMPLEX **for** cgemv, scgemv

DOUBLE COMPLEX **for** zgemv, dzgemv

Specifies the scalar *beta*. When *beta* is set to zero, then *y* need not be set on input.

y REAL **for** sgemv

DOUBLE PRECISION **for** dgemv

COMPLEX **for** cgemv, scgemv

DOUBLE COMPLEX for zgemv, dzgemv
 Array, size at least $(1 + (m - 1) * \text{abs}(incy))$ when $\text{trans} = 'N'$ or ' n '
 and at least $(1 + (n - 1) * \text{abs}(incy))$ otherwise. Before entry with non-zero β , the incremented array y must contain the vector y .

$incy$ INTEGER. Specifies the increment for the elements of y .
 The value of $incy$ must not be zero.

Output Parameters

y Updated vector y .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gemv` interface are the following:

a	Holds the matrix A of size (m,n) .
x	Holds the vector with the number of elements rx where $rx = n$ if $\text{trans} = 'N'$, $rx = m$ otherwise.
y	Holds the vector with the number of elements ry where $ry = m$ if $\text{trans} = 'N'$, $ry = n$ otherwise.
$trans$	Must be ' N ', ' C ', or ' T '. The default value is ' N '.
$alpha$	The default value is 1.
$beta$	The default value is 0.

?ger

Performs a rank-1 update of a general matrix.

Syntax

FORTRAN 77:

```
call sger(m, n, alpha, x, incx, y, incy, a, lda)
call dger(m, n, alpha, x, incx, y, incy, a, lda)
```

Fortran 95:

```
call ger(a, x, y [,alpha])
```

C:

```
void cblas_sger (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const
float alpha, const float *x, const MKL_INT incx, const float *y, const MKL_INT incy,
float *a, const MKL_INT lda);
```

```
void cblas_dger (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const
double alpha, const double *x, const MKL_INT incx, const double *y, const MKL_INT
incy, double *a, const MKL_INT lda);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?ger routines perform a matrix-vector operation defined as

```
A := alpha*x*y' + A,
```

where:

alpha is a scalar,
x is an *m*-element vector,
y is an *n*-element vector,
A is an *m*-by-*n* general matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for sger DOUBLE PRECISION for dger Specifies the scalar <i>alpha</i> .
<i>x</i>	REAL for sger DOUBLE PRECISION for dger Array, size at least $(1 + (m - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>m</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	REAL for sger DOUBLE PRECISION for dger Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> .

The value of *incy* must not be zero.

<i>a</i>	REAL for sger DOUBLE PRECISION for dger Array, size (<i>lda</i> , <i>n</i>). Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix of coefficients.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, m)$.

Output Parameters

<i>a</i>	Overwritten by the updated matrix.
----------	------------------------------------

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *ger* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m,n</i>).
<i>x</i>	Holds the vector with the number of elements <i>m</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .
<i>alpha</i>	The default value is 1.

?gerc

Performs a rank-1 update (conjugated) of a general matrix.

Syntax

FORTRAN 77:

```
call cgerc(m, n, alpha, x, incx, y, incy, a, lda)
call zgerc(m, n, alpha, x, incx, y, incy, a, lda)
```

Fortran 95:

```
call gerc(a, x, y [,alpha])
```

C:

```
void cblas_cgerc (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *a, const MKL_INT lda);
void cblas_zgerc (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *a, const MKL_INT lda);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?gerc routines perform a matrix-vector operation defined as

```
A := alpha*x*conjg(y') + A,
```

where:

alpha is a scalar,
x is an *m*-element vector,
y is an *n*-element vector,
A is an *m*-by-*n* matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for cgerc DOUBLE COMPLEX for zgerc Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for cgerc DOUBLE COMPLEX for zgerc Array, size at least $(1 + (m - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>m</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	COMPLEX for cgerc DOUBLE COMPLEX for zgerc Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>a</i>	COMPLEX for cgerc

DOUBLE COMPLEX for zgerc

Array, size (*lda, n*).

Before entry, the leading m -by- n part of the array *a* must contain the matrix of coefficients.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least $\max(1, m)$.

Output Parameters

a

Overwritten by the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gerc* interface are the following:

a

Holds the matrix *A* of size (*m,n*).

x

Holds the vector with the number of elements *m*.

y

Holds the vector with the number of elements *n*.

alpha

The default value is 1.

?geru

Performs a rank-1 update (unconjugated) of a general matrix.

Syntax

FORTRAN 77:

```
call cgeru(m, n, alpha, x, incx, y, incy, a, lda)
call zgeru(m, n, alpha, x, incx, y, incy, a, lda)
```

Fortran 95:

```
call geru(a, x, y [,alpha])
```

C:

```
void cblas_cgeru (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *a, const MKL_INT lda);
void cblas_zgeru (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *a, const MKL_INT lda);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90

- C: mkl.h

Description

The ?geru routines perform a matrix-vector operation defined as

$$A := \alpha * x * y' + A,$$

where:

α is a scalar,
 x is an m -element vector,
 y is an n -element vector,
 A is an m -by- n matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>m</i>	INTEGER. Specifies the number of rows of the matrix A . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix A . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for cgeru DOUBLE COMPLEX for zgeru Specifies the scalar α .
<i>x</i>	COMPLEX for cgeru DOUBLE COMPLEX for zgeru Array, size at least $(1 + (m - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the m -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	COMPLEX for cgeru DOUBLE COMPLEX for zgeru Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the n -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>a</i>	COMPLEX for cgeru DOUBLE COMPLEX for zgeru Array, size (lda, n) .

Before entry, the leading m -by- n part of the array a must contain the matrix of coefficients.

lda INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of lda must be at least $\max(1, m)$.

Output Parameters

a Overwritten by the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geru` interface are the following:

a	Holds the matrix A of size (m,n) .
x	Holds the vector with the number of elements m .
y	Holds the vector with the number of elements n .
$alpha$	The default value is 1.

?hbmv

Computes a matrix-vector product using a Hermitian band matrix.

Syntax

FORTRAN 77:

```
call chbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
call zhbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
```

Fortran 95:

```
call hbmv(a, x, y [,uplo][,alpha] [,beta])
```

C:

```
void cblas_chbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *x,
const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);

void cblas_zhbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *x,
const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `blas.f90`
- C: `mkl.h`

Description

The `?hbmv` routines perform a matrix-vector operation defined as $y := \alpha A^*x + \beta y$,

where:

alpha and *beta* are scalars,
x and *y* are *n*-element vectors,
A is an *n*-by-*n* Hermitian band matrix, with *k* super-diagonals.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian band matrix <i>A</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. Specifies the number of super-diagonals of the matrix <i>A</i> . The value of <i>k</i> must satisfy $0 \leq k$.
<i>alpha</i>	COMPLEX for chbmv DOUBLE COMPLEX for zhbmv Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for chbmv DOUBLE COMPLEX for zhbmv Array, size (<i>lda</i> , <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading $(k + 1)$ by <i>n</i> part of the array <i>a</i> must contain the upper triangular band part of the Hermitian matrix. The matrix must be supplied column-by-column, with the leading diagonal of the matrix in row $(k + 1)$ of the array, the first super-diagonal starting at position 2 in row <i>k</i> , and so on. The top left <i>k</i> by <i>k</i> triangle of the array <i>a</i> is not referenced. The following program segment transfers the upper triangular part of a Hermitian band matrix from conventional full matrix storage (<i>matrix</i>) to band storage (<i>a</i>):

```

      do 20, j = 1, n
        m = k + 1 - j
        do 10, i = max( 1, j - k ), j
          a( m + i, j ) = matrix( i, j )
10      continue
20      continue

```

Before entry with *uplo* = 'L' or 'l', the leading $(k + 1)$ by *n* part of the array *a* must contain the lower triangular band part of the Hermitian matrix, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers the lower triangular part of a Hermitian band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```

do 20, j = 1, n
    m = 1 - j
    do 10, i = j, min( n, j + k )
        a( m + i, j ) = matrix( i, j )
10         continue
20         continue

```

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least $(k + 1)$.

x COMPLEX for chbmv

DOUBLE COMPLEX for zhbmv

Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array *x* must contain the vector *x*.

incx INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

beta COMPLEX for chbmv

DOUBLE COMPLEX for zhbmv

Specifies the scalar *beta*.

y COMPLEX for chbmv

DOUBLE COMPLEX for zhbmv

Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array *y* must contain the vector *y*.

incy INTEGER. Specifies the increment for the elements of *y*.

The value of *incy* must not be zero.

Output Parameters

y Overwritten by the updated vector *y*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *hbmv* interface are the following:

a Holds the array *a* of size $(k+1,n)$.

x Holds the vector with the number of elements *n*.

y Holds the vector with the number of elements *n*.

<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?hemv

Computes a matrix-vector product using a Hermitian matrix.

Syntax

FORTRAN 77:

```
call chemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call zhemv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
```

Fortran 95:

```
call hemv(a, x, y [,uplo] [,alpha] [,beta])
```

C:

```
void cblas_chemv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT
incx, const void *beta, void *y, const MKL_INT incy);

void cblas_zhemv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT
incx, const void *beta, void *y, const MKL_INT incy);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?hemv routines perform a matrix-vector operation defined as

```
y := alpha*A*x + beta*y,
```

where:

alpha and *beta* are scalars,
x and *y* are *n*-element vectors,
A is an *n*-by-*n* Hermitian matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular of the array <i>a</i> is used.

If $uplo = 'L'$ or $'l'$, then the low triangular of the array a is used.

n INTEGER. Specifies the order of the matrix A . The value of n must be at least zero.

α COMPLEX for chemv

DOUBLE COMPLEX for zhemv

Specifies the scalar α .

a COMPLEX for chemv

DOUBLE COMPLEX for zhemv

Array, size (lda, n) .

Before entry with $uplo = 'U'$ or $'u'$, the leading n -by- n upper triangular part of the array a must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of a is not referenced. Before entry with $uplo = 'L'$ or $'l'$, the leading n -by- n lower triangular part of the array a must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of a is not referenced.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

lda INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of lda must be at least $\max(1, n)$.

x COMPLEX for chemv

DOUBLE COMPLEX for zhemv

Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the n -element vector x .

$incx$ INTEGER. Specifies the increment for the elements of x .

The value of $incx$ must not be zero.

β COMPLEX for chemv

DOUBLE COMPLEX for zhemv

Specifies the scalar β . When β is supplied as zero then y need not be set on input.

y COMPLEX for chemv

DOUBLE COMPLEX for zhemv

Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array y must contain the n -element vector y .

$incy$ INTEGER. Specifies the increment for the elements of y .

The value of $incy$ must not be zero.

Output Parameters

y Overwritten by the updated vector y .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hemv` interface are the following:

<code>a</code>	Holds the matrix A of size (n,n) .
<code>x</code>	Holds the vector with the number of elements n .
<code>y</code>	Holds the vector with the number of elements n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>alpha</code>	The default value is 1.
<code>beta</code>	The default value is 0.

?her

Performs a rank-1 update of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call cher(uplo, n, alpha, x, incx, a, lda)
call zher(uplo, n, alpha, x, incx, a, lda)
```

Fortran 95:

```
call her(a, x [,uplo] [, alpha])
```

C:

```
void cblas_cher (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const void *x, const MKL_INT incx, void *a, const MKL_INT lda);
void cblas_zher (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const void *x, const MKL_INT incx, void *a, const MKL_INT lda);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `blas.f90`
- C: `mkl.h`

Description

The `?her` routines perform a matrix-vector operation defined as

```
A := alpha*x*conjg(x') + A,
```

where:

`alpha` is a real scalar,

`x` is an n -element vector,

`A` is an n -by- n Hermitian matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for cher DOUBLE PRECISION for zher Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for cher DOUBLE COMPLEX for zher Array, dimension at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>a</i>	COMPLEX for cher DOUBLE COMPLEX for zher Array, size (<i>lda</i> , <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>a</i> is not referenced. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = 'L' or 'l', the lower triangular part of the array <i>a</i> is overwritten by the lower triangular part of the updated matrix. The imaginary parts of the diagonal elements are set to zero.
----------	---

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `her` interface are the following:

<code>a</code>	Holds the matrix A of size (n,n) .
<code>x</code>	Holds the vector with the number of elements n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>alpha</code>	The default value is 1.

?her2

Performs a rank-2 update of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call cher2(uplo, n, alpha, x, incx, y, incy, a, lda)
call zher2(uplo, n, alpha, x, incx, y, incy, a, lda)
```

Fortran 95:

```
call her2(a, x, y [,uplo][,alpha])
```

C:

```
void cblas_cher2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT
incy, void *a, const MKL_INT lda);

void cblas_zher2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT
incy, void *a, const MKL_INT lda);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `blas.f90`
- C: `mkl.h`

Description

The `?her2` routines perform a matrix-vector operation defined as

$$A := \alpha *x*\text{conjg}(y') + \text{conjg}(\alpha)*y * \text{conjg}(x') + A,$$

where:

`alpha` is a scalar,

`x` and `y` are n -element vectors,

`A` is an n -by- n Hermitian matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for cher2 DOUBLE COMPLEX for zher2 Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for cher2 DOUBLE COMPLEX for zher2 Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	COMPLEX for cher2 DOUBLE COMPLEX for zher2 Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>a</i>	COMPLEX for cher2 DOUBLE COMPLEX for zher2 Array, size (lda, n) . Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>a</i> is not referenced. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

- a*
- With *uplo* = 'U' or 'u', the upper triangular part of the array *a* is overwritten by the upper triangular part of the updated matrix.
- With *uplo* = 'L' or 'l', the lower triangular part of the array *a* is overwritten by the lower triangular part of the updated matrix.
- The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `her2` interface are the following:

- | | |
|--------------|---|
| <i>a</i> | Holds the matrix <i>A</i> of size (<i>n,n</i>). |
| <i>x</i> | Holds the vector with the number of elements <i>n</i> . |
| <i>y</i> | Holds the vector with the number of elements <i>n</i> . |
| <i>uplo</i> | Must be 'U' or 'L'. The default value is 'U'. |
| <i>alpha</i> | The default value is 1. |

?hpmv

Computes a matrix-vector product using a Hermitian packed matrix.

Syntax

FORTRAN 77:

```
call chpmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
call zhpmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
```

Fortran 95:

```
call hpmv(ap, x, y [,uplo][,alpha][,beta])
```

C:

```
void cblas_chpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *ap, const void *x, const MKL_INT incx, const void
*beta, void *y, const MKL_INT incy);

void cblas_zhpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *ap, const void *x, const MKL_INT incx, const void
*beta, void *y, const MKL_INT incy);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `blas.f90`
- C: `mkl.h`

Description

The ?hpmv routines perform a matrix-vector operation defined as

```
y := alpha*A*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> .
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for chpmv DOUBLE COMPLEX for zhpmv Specifies the scalar <i>alpha</i> .
<i>ap</i>	COMPLEX for chpmv DOUBLE COMPLEX for zhpmv Array, size at least ((<i>n</i> *(<i>n</i> + 1))/2). Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (1, 2) and <i>a</i> (2, 2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1, 1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (2, 1) and <i>a</i> (3, 1) respectively, and so on. The imaginary parts of the diagonal elements need not be set and are assumed to be zero.
<i>x</i>	COMPLEX for chpmv DOUBLE PRECISION COMPLEX for zhpmv Array, size at least (1 + (<i>n</i> - 1)*abs(<i>incx</i>)). Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>beta</i>	COMPLEX for chpmv

```
void cblas_zhpr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const void *x, const MKL_INT incx, void *ap);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?hpr routines perform a matrix-vector operation defined as

```
A := alpha*x*conjg(x') + A,
```

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'U' or 'u', the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'L' or 'l', the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> .
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for chpr DOUBLE PRECISION for zhpr Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for chpr DOUBLE COMPLEX for zhpr Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . <i>incx</i> must not be zero.
<i>ap</i>	COMPLEX for chpr DOUBLE COMPLEX for zhpr Array, size at least $((n * (n + 1)) / 2)$.

Before entry with $\text{uplo} = \text{'U'}$ or 'u' , the array ap must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that $\text{ap}(1)$ contains $a(1, 1)$, $\text{ap}(2)$ and $\text{ap}(3)$ contain $a(1, 2)$ and $a(2, 2)$ respectively, and so on.

Before entry with $\text{uplo} = \text{'L'}$ or 'l' , the array ap must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that $\text{ap}(1)$ contains $a(1, 1)$, $\text{ap}(2)$ and $\text{ap}(3)$ contain $a(2, 1)$ and $a(3, 1)$ respectively, and so on.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

ap

With $\text{uplo} = \text{'U'}$ or 'u' , overwritten by the upper triangular part of the updated matrix.

With $\text{uplo} = \text{'L'}$ or 'l' , overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpr` interface are the following:

ap

Holds the array ap of size $(n*(n+1)/2)$.

x

Holds the vector with the number of elements n .

uplo

Must be 'U' or 'L' . The default value is 'U' .

alpha

The default value is 1.

?hpr2

Performs a rank-2 update of a Hermitian packed matrix.

Syntax

FORTRAN 77:

```
call chpr2(uplo, n, alpha, x, incx, y, incy, ap)
call zhpr2(uplo, n, alpha, x, incx, y, incy, ap)
```

Fortran 95:

```
call hpr2(ap, x, y [,uplo][,alpha])
```

C:

```
void cblas_chpr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT
incy, void *ap);
```

```
void cblas_zhpr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT
incy, void *ap);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?hpr2 routines perform a matrix-vector operation defined as

```
A := alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A,
```

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* Hermitian matrix, supplied in packed form.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> .
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for chpr2 DOUBLE COMPLEX for zhpr2 Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for chpr2 DOUBLE COMPLEX for zhpr2 Array, dimension at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	COMPLEX for chpr2 DOUBLE COMPLEX for zhpr2

Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array y must contain the n -element vector y .

incy INTEGER. Specifies the increment for the elements of y .

The value of *incy* must not be zero.

ap COMPLEX for chpr2

DOUBLE COMPLEX for zhpr2

Array, size at least $((n * (n + 1)) / 2)$.

Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains $a(1, 1)$, *ap*(2) and *ap*(3) contain $a(1, 2)$ and $a(2, 2)$ respectively, and so on.

Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that *ap*(1) contains $a(1, 1)$, *ap*(2) and *ap*(3) contain $a(2, 1)$ and $a(3, 1)$ respectively, and so on.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

Output Parameters

ap With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements need be set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpr2` interface are the following:

ap Holds the array *ap* of size $(n * (n + 1) / 2)$.

x Holds the vector with the number of elements *n*.

y Holds the vector with the number of elements *n*.

uplo Must be 'U' or 'L'. The default value is 'U'.

alpha The default value is 1.

?sbmv

Computes a matrix-vector product using a symmetric band matrix.

Syntax

FORTRAN 77:

```
call ssbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
call dsbmv(uplo, n, k, alpha, a, lda, x, incx, beta, y, incy)
```

Fortran 95:

```
call sbmv(a, x, y [,uplo][,alpha] [,beta])
```

C:

```
void cblas_ssbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const MKL_INT k, const float alpha, const float *a, const MKL_INT lda, const float *x,
const MKL_INT incx, const float beta, float *y, const MKL_INT incy);

void cblas_dsbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const MKL_INT k, const double alpha, const double *a, const MKL_INT lda, const double
*x, const MKL_INT incx, const double beta, double *y, const MKL_INT incy);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?sbmv routines perform a matrix-vector operation defined as

$$y := \alpha A^* x + \beta y,$$

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric band matrix, with *k* super-diagonals.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix <i>A</i> is used: if <i>uplo</i> = 'U' or 'u' - upper triangular part; if <i>uplo</i> = 'L' or 'l' - low triangular part.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. Specifies the number of super-diagonals of the matrix <i>A</i> . The value of <i>k</i> must satisfy $0 \leq k$.
<i>alpha</i>	REAL for ssbmv DOUBLE PRECISION for dsbmv Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for ssbmv

DOUBLE PRECISION for dsbmv

Array, size (lda, n). Before entry with $uplo = 'U'$ or ' u ', the leading $(k + 1)$ by n part of the array a must contain the upper triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row $(k + 1)$ of the array, the first super-diagonal starting at position 2 in row k , and so on. The top left k by k triangle of the array a is not referenced.

The following program segment transfers the upper triangular part of a symmetric band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
do 20, j = 1, n
      m = k + 1 - j
      do 10, i = max( 1, j - k ), j
            a( m + i, j ) = matrix( i, j )
10          continue
20          continue
```

Before entry with $uplo = 'L'$ or ' l ', the leading $(k + 1)$ by n part of the array a must contain the lower triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right k by k triangle of the array a is not referenced.

The following program segment transfers the lower triangular part of a symmetric band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
do 20, j = 1, n
      m = 1 - j
      do 10, i = j, min( n, j + k )
            a( m + i, j ) = matrix( i, j )
10          continue
20          continue
```

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least $(k + 1)$.

x

REAL for ssbmv

DOUBLE PRECISION for dsbmv

Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array *x* must contain the vector *x*.

incx

INTEGER. Specifies the increment for the elements of *x*.

The value of *incx* must not be zero.

beta

REAL for ssbmv

DOUBLE PRECISION for dsbmv

Specifies the scalar *beta*.

y

REAL for ssbmv

DOUBLE PRECISION for dsbmv

Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array y must contain the vector y .

$incy$ INTEGER. Specifies the increment for the elements of y .
The value of $incy$ must not be zero.

Output Parameters

y Overwritten by the updated vector y .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbmv` interface are the following:

a	Holds the array a of size $(k+1,n)$.
x	Holds the vector with the number of elements n .
y	Holds the vector with the number of elements n .
$uplo$	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
$alpha$	The default value is 1.
$beta$	The default value is 0.

?spmv

Computes a matrix-vector product using a symmetric packed matrix.

Syntax

FORTRAN 77:

```
call sspmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
call dspmv(uplo, n, alpha, ap, x, incx, beta, y, incy)
```

Fortran 95:

```
call spmv(ap, x, y [,uplo][,alpha] [,beta])
```

C:

```
void cblas_sspmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *ap, const float *x, const MKL_INT incx, const float
beta, float *y, const MKL_INT incy);

void cblas_dspmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *ap, const double *x, const MKL_INT incx, const double
beta, double *y, const MKL_INT incy);
```

Include Files

- Fortran: `mkl.fi`

- Fortran 95: blas.f90
- C: mkl.h

Description

The ?spmv routines perform a matrix-vector operation defined as

```
y := alpha*A*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> .
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for sspmv DOUBLE PRECISION for dspmv Specifies the scalar <i>alpha</i> .
<i>ap</i>	REAL for sspmv DOUBLE PRECISION for dspmv Array, size at least ((n*(n + 1))/2). Before entry with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1,1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (1,2) and <i>a</i> (2, 2) respectively, and so on. Before entry with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains <i>a</i> (1,1), <i>ap</i> (2) and <i>ap</i> (3) contain <i>a</i> (2,1) and <i>a</i> (3,1) respectively, and so on.
<i>x</i>	REAL for sspmv DOUBLE PRECISION for dspmv Array, size at least (1 + (n - 1)*abs(<i>incx</i>)). Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

The value of *incx* must not be zero.

<i>beta</i>	REAL for sspmv DOUBLE PRECISION for dspmv Specifies the scalar <i>beta</i> . When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.
<i>y</i>	REAL for sspmv DOUBLE PRECISION for dspmv Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated vector <i>y</i> .
----------	--

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spmv` interface are the following:

<i>ap</i>	Holds the array <i>ap</i> of size $(n*(n+1)/2)$.
<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?spr

Performs a rank-1 update of a symmetric packed matrix.

Syntax

FORTRAN 77:

```
call sspr(uplo, n, alpha, x, incx, ap)
call dspr(uplo, n, alpha, x, incx, ap)
```

Fortran 95:

```
call spr(ap, x [,uplo] [, alpha])
```

C:

```
void cblas_sspr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *x, const MKL_INT incx, float *ap);

void cblas_dspr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *x, const MKL_INT incx, double *ap);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?spr routines perform a matrix-vector operation defined as

```
a := alpha*x*x' + A,
```

where:

alpha is a real scalar,

x is an *n*-element vector,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> .
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for sspr DOUBLE PRECISION for dspr Specifies the scalar <i>alpha</i> .
<i>x</i>	REAL for sspr DOUBLE PRECISION for dspr Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>ap</i>	REAL for sspr DOUBLE PRECISION for dspr

Before entry with `uplo = 'U'` or `'u'`, the array `ap` must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that `ap(1)` contains $a(1, 1)$, `ap(2)` and `ap(3)` contain $a(1, 2)$ and $a(2, 2)$ respectively, and so on.

Before entry with `uplo = 'L'` or `'l'`, the array `ap` must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that `ap(1)` contains $a(1, 1)$, `ap(2)` and `ap(3)` contain $a(2, 1)$ and $a(3, 1)$ respectively, and so on.

Output Parameters

`ap`

With `uplo = 'U'` or `'u'`, overwritten by the upper triangular part of the updated matrix.

With `uplo = 'L'` or `'l'`, overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spr` interface are the following:

`ap` Holds the array `ap` of size $(n*(n+1)/2)$.

`x` Holds the vector with the number of elements n .

`uplo` Must be `'U'` or `'L'`. The default value is `'U'`.

`alpha` The default value is 1.

?spr2

Performs a rank-2 update of a symmetric packed matrix.

Syntax

FORTRAN 77:

```
call sspr2(uplo, n, alpha, x, incx, y, incy, ap)
call dspr2(uplo, n, alpha, x, incx, y, incy, ap)
```

Fortran 95:

```
call spr2(ap, x, y [,uplo][,alpha])
```

C:

```
void cblas_sspr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *x, const MKL_INT incx, const float *y, const MKL_INT
incy, float *ap);

void cblas.dspr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *x, const MKL_INT incx, const double *y, const MKL_INT
incy, double *ap);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?spr2 routines perform a matrix-vector operation defined as

```
A := alpha*x*y' + alpha*y*x' + A,
```

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix, supplied in packed form.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'L' or 'l', then the low triangular part of the matrix <i>A</i> is supplied in the packed array <i>ap</i> .
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for sspr2 DOUBLE PRECISION for dspr2 Specifies the scalar <i>alpha</i> .
<i>x</i>	REAL for sspr2 DOUBLE PRECISION for dspr2 Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	REAL for sspr2 DOUBLE PRECISION for dspr2 Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.

ap REAL for sspr2
 DOUBLE PRECISION for dspr2

Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains $a(1, 1)$, *ap*(2) and *ap*(3) contain $a(1, 2)$ and $a(2, 2)$ respectively, and so on.

Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that *ap*(1) contains $a(1, 1)$, *ap*(2) and *ap*(3) contain $a(2, 1)$ and $a(3, 1)$ respectively, and so on.

Output Parameters

ap With *uplo* = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.
 With *uplo* = 'L' or 'l', overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *spr2* interface are the following:

<i>ap</i>	Holds the array <i>ap</i> of size $(n*(n+1)/2)$.
<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>y</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.

?symv

Computes a matrix-vector product for a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call dsymv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
```

Fortran 95:

```
call symv(a, x, y [,uplo] [,alpha] [,beta])
```

C:

```
void cblas_ssymv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *a, const MKL_INT lda, const float *x, const MKL_INT incx,
const float beta, float *y, const MKL_INT incy);
```

```
void cblas_dsymv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *a, const MKL_INT lda, const double *x, const MKL_INT incx,
const double beta, double *y, const MKL_INT incy);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?symv routines perform a matrix-vector operation defined as

```
y := alpha*A*x + beta*y,
```

where:

alpha and *beta* are scalars,
x and *y* are *n*-element vectors,
A is an *n*-by-*n* symmetric matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for ssymv DOUBLE PRECISION for dsymv Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for ssymv DOUBLE PRECISION for dsymv Array, size (<i>lda</i> , <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix <i>A</i> and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix <i>A</i> and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least <code>max(1, n)</code> .
<i>x</i>	REAL for ssymv

DOUBLE PRECISION for `dsymv`
 Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the n -element vector x .

incx INTEGER. Specifies the increment for the elements of x .
 The value of *incx* must not be zero.

beta REAL for `ssymv`
 DOUBLE PRECISION for `dsymv`
 Specifies the scalar *beta*.
 When *beta* is supplied as zero, then y need not be set on input.

y REAL for `ssymv`
 DOUBLE PRECISION for `dsymv`
 Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array y must contain the n -element vector y .

incy INTEGER. Specifies the increment for the elements of y .
 The value of *incy* must not be zero.

Output Parameters

y Overwritten by the updated vector y .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `symv` interface are the following:

<i>a</i>	Holds the matrix A of size (n,n) .
<i>x</i>	Holds the vector with the number of elements n .
<i>y</i>	Holds the vector with the number of elements n .
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?syr

Performs a rank-1 update of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssyr(uplo, n, alpha, x, incx, a, lda)
call dsyr(uplo, n, alpha, x, incx, a, lda)
```

Fortran 95:

```
call syr(a, x [,uplo] [, alpha])
```

C:

```
void cblas_ssyr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *x, const MKL_INT incx, float *a, const MKL_INT lda);
void cblas_dsy (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *x, const MKL_INT incx, double *a, const MKL_INT lda);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?syr routines perform a matrix-vector operation defined as

$$A := \alpha x x' + A,$$

where:

α is a real scalar,

x is an n -element vector,

A is an n -by- n symmetric matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for ssyr DOUBLE PRECISION for dsy Specifies the scalar α .
<i>x</i>	REAL for ssyr DOUBLE PRECISION for dsy Array, size at least $(1 + (n-1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the n -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>a</i>	REAL for ssyr

DOUBLE PRECISION for `dsyr`

Array, size (`lda, n`).

Before entry with `uplo = 'U'` or `'u'`, the leading n -by- n upper triangular part of the array `a` must contain the upper triangular part of the symmetric matrix A and the strictly lower triangular part of `a` is not referenced.

Before entry with `uplo = 'L'` or `'l'`, the leading n -by- n lower triangular part of the array `a` must contain the lower triangular part of the symmetric matrix A and the strictly upper triangular part of `a` is not referenced.

`lda`

INTEGER. Specifies the leading dimension of `a` as declared in the calling (sub)program. The value of `lda` must be at least `max(1, n)`.

Output Parameters

`a`

With `uplo = 'U'` or `'u'`, the upper triangular part of the array `a` is overwritten by the upper triangular part of the updated matrix.

With `uplo = 'L'` or `'l'`, the lower triangular part of the array `a` is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `syr` interface are the following:

`a` Holds the matrix A of size (n,n) .

`x` Holds the vector with the number of elements n .

`uplo` Must be `'U'` or `'L'`. The default value is `'U'`.

`alpha` The default value is 1.

?syr2

Performs a rank-2 update of symmetric matrix.

Syntax

FORTRAN 77:

```
call ssyr2(uplo, n, alpha, x, incx, y, incy, a, lda)
call dsyr2(uplo, n, alpha, x, incx, y, incy, a, lda)
```

Fortran 95:

```
call syr2(a, x, y [,uplo] [,alpha])
```

C:

```
void cblas_ssyr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const float alpha, const float *x, const MKL_INT incx, const float *y, const MKL_INT
incy, float *a, const MKL_INT lda);
```

```
void cblas_dsy2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const double alpha, const double *x, const MKL_INT incx, const double *y, const MKL_INT incy,
double *a, const MKL_INT lda);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?syr2 routines perform a matrix-vector operation defined as

```
A := alpha*x*y' + alpha*y*x' + A,
```

where:

alpha is a scalar,

x and *y* are *n*-element vectors,

A is an *n*-by-*n* symmetric matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for ssyr2 DOUBLE PRECISION for dsyr2 Specifies the scalar <i>alpha</i> .
<i>x</i>	REAL for ssyr2 DOUBLE PRECISION for dsyr2 Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	REAL for ssyr2 DOUBLE PRECISION for dsyr2 Array, size at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array <i>y</i> must contain the <i>n</i> -element vector <i>y</i> .

<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must not be zero.
<i>a</i>	REAL for ssyr2 DOUBLE PRECISION for dsyr2 Array, size (<i>lda</i> , <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated matrix. With <i>uplo</i> = 'L' or 'l', the lower triangular part of the array <i>a</i> is overwritten by the lower triangular part of the updated matrix.
----------	--

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *syr2* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n,n</i>).
<i>x</i>	Holds the vector <i>x</i> of length <i>n</i> .
<i>y</i>	Holds the vector <i>y</i> of length <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>alpha</i>	The default value is 1.

?tbmv

Computes a matrix-vector product using a triangular band matrix.

Syntax

FORTRAN 77:

```
call stbmv(uplo, trans, diag, n, k, a, lda, x, incx)
call dtbmv(uplo, trans, diag, n, k, a, lda, x, incx)
call ctbmv(uplo, trans, diag, n, k, a, lda, x, incx)
call ztbmv(uplo, trans, diag, n, k, a, lda, x, incx)
```

Fortran 95:

```
call tbmv(a, x [,uplo] [, trans] [,diag])
```

C:

```
void cblas_stbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
float *a, const MKL_INT lda, float *x, const MKL_INT incx);

void cblas_dtbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
double *a, const MKL_INT lda, double *x, const MKL_INT incx);

void cblas_ctbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
void *a, const MKL_INT lda, void *x, const MKL_INT incx);

void cblas_ztbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
void *a, const MKL_INT lda, void *x, const MKL_INT incx);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?tbmv routines perform one of the matrix-vector operations defined as

$$x := A^*x, \text{ or } x := A'^*x, \text{ or } x := \text{conjg}(A')^*x,$$

where:

x is an n -element vector,

A is an n -by- n unit, or non-unit, upper or lower triangular band matrix, with $(k + 1)$ diagonals.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is an upper or lower triangular matrix: $uplo = 'U'$ or ' u ' if $uplo = 'L'$ or ' l ', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the operation: if $trans= 'N'$ or ' n ', then $x := A^*x$; if $trans= 'T'$ or ' t ', then $x := A'^*x$; if $trans= 'C'$ or ' c ', then $x := \text{conjg}(A')^*x$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix A is unit triangular: if $uplo = 'U'$ or ' u ' then the matrix is unit triangular;

if $\text{diag} = \text{'N'}$ or 'n' , then the matrix is not unit triangular.

n INTEGER. Specifies the order of the matrix A . The value of n must be at least zero.

k INTEGER. On entry with $\text{uplo} = \text{'U'}$ or 'u' specifies the number of super-diagonals of the matrix A . On entry with $\text{uplo} = \text{'L'}$ or 'l' , k specifies the number of sub-diagonals of the matrix a .

The value of k must satisfy $0 \leq k$.

a REAL for stbmv

DOUBLE PRECISION for dtbmv

COMPLEX for ctbmv

DOUBLE COMPLEX for ztbmv

Array, size (lda, n) .

Before entry with $\text{uplo} = \text{'U'}$ or 'u' , the leading $(k + 1)$ by n part of the array a must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row $(k + 1)$ of the array, the first super-diagonal starting at position 2 in row k , and so on. The top left k by k triangle of the array a is not referenced. The following program segment transfers an upper triangular band matrix from conventional full matrix storage (matrix) to band storage (a):

```
do 20, j = 1, n
      m = k + 1 - j
      do 10, i = max( 1, j - k ), j
            a( m + i, j ) = matrix( i, j )
10          continue
20          continue
```

Before entry with $\text{uplo} = \text{'L'}$ or 'l' , the leading $(k + 1)$ by n part of the array a must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right k by k triangle of the array a is not referenced. The following program segment transfers a lower triangular band matrix from conventional full matrix storage (matrix) to band storage (a):

```
do 20, j = 1, n
      m = 1 - j
      do 10, i = j, min( n, j + k )
            a( m + i, j ) = matrix( i, j )
10          continue
20          continue
```

Note that when $\text{uplo} = \text{'U'}$ or 'u' , the elements of the array a corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of lda must be at least $(k + 1)$.

x REAL for stbmv

DOUBLE PRECISION for dtbmv
 COMPLEX for ctbmv
 DOUBLE COMPLEX for ztbmv
 Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the n -element vector x .

$incx$ INTEGER. Specifies the increment for the elements of x .
 The value of $incx$ must not be zero.

Output Parameters

x Overwritten with the transformed vector x .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tbsv` interface are the following:

a	Holds the array a of size $(k+1,n)$.
x	Holds the vector with the number of elements n .
$uplo$	Must be 'U' or 'L'. The default value is 'U'.
$trans$	Must be 'N', 'C', or 'T'. The default value is 'N'.
$diag$	Must be 'N' or 'U'. The default value is 'N'.

?tbsv

Solves a system of linear equations whose coefficients are in a triangular band matrix.

Syntax

FORTRAN 77:

```
call stbsv(uplo, trans, diag, n, k, a, lda, x, incx)
call dtbsv(uplo, trans, diag, n, k, a, lda, x, incx)
call ctbsv(uplo, trans, diag, n, k, a, lda, x, incx)
call ztbsv(uplo, trans, diag, n, k, a, lda, x, incx)
```

Fortran 95:

```
call tbsv(a, x [,uplo] [, trans] [,diag])
```

C:

```
void cblas_stbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
float *a, const MKL_INT lda, float *x, const MKL_INT incx);
```

```

void cblas_dtbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
double *a, const MKL_INT lda, double *x, const MKL_INT incx);

void cblas_ctbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
void *a, const MKL_INT lda, void *x, const MKL_INT incx);

void cblas_ztbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const
void *a, const MKL_INT lda, void *x, const MKL_INT incx);

```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?tbsv routines solve one of the following systems of equations:

$$A*x = b, \text{ or } A'*x = b, \text{ or } \text{conjg}(A')*x = b,$$

where:

b and *x* are *n*-element vectors,

A is an *n*-by-*n* unit, or non-unit, upper or lower triangular band matrix, with (*k* + 1) diagonals.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is an upper or lower triangular matrix: if <i>uplo</i> = 'U' or 'u' the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the system of equations: if <i>trans</i> = 'N' or 'n', then $A*x = b$; if <i>trans</i> = 'T' or 't', then $A'*x = b$; if <i>trans</i> = 'C' or 'c', then $\text{conjg}(A')*x = b$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.

k

INTEGER. On entry with *uplo* = 'U' or 'u', *k* specifies the number of super-diagonals of the matrix *A*. On entry with *uplo* = 'L' or 'l', *k* specifies the number of sub-diagonals of the matrix *A*.

The value of *k* must satisfy $0 \leq k$.

a

REAL for stbsv
 DOUBLE PRECISION for dtbsv
 COMPLEX for ctbsv
 DOUBLE COMPLEX for ztbsv

Array, size (*lda*, *n*).

Before entry with *uplo* = 'U' or 'u', the leading $(k + 1)$ by *n* part of the array *a* must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row $(k + 1)$ of the array, the first super-diagonal starting at position 2 in row *k*, and so on. The top left *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers an upper triangular band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
do 20, j = 1, n
      m = k + 1 - j
      do 10, i = max( 1, j - k ), j
            a( m + i, j ) = matrix( i, j )
10      continue
20      continue
```

Before entry with *uplo* = 'L' or 'l', the leading $(k + 1)$ by *n* part of the array *a* must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 1 of the array, the first sub-diagonal starting at position 1 in row 2, and so on. The bottom right *k* by *k* triangle of the array *a* is not referenced.

The following program segment transfers a lower triangular band matrix from conventional full matrix storage (*matrix*) to band storage (*a*):

```
do 20, j = 1, n
      m = 1 - j
      do 10, i = j, min( n, j + k )
            a( m + i, j ) = matrix( i, j )
10      continue
20      continue
```

When *diag* = 'U' or 'u', the elements of the array *a* corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least $(k + 1)$.

x

REAL for stbsv
 DOUBLE PRECISION for dtbsv
 COMPLEX for ctbsv

DOUBLE COMPLEX for `ztbsv`
 Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the n -element right-hand side vector b .

`incx` INTEGER. Specifies the increment for the elements of x .
 The value of `incx` must not be zero.

Output Parameters

`x` Overwritten with the solution vector x .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tbsv` interface are the following:

<code>a</code>	Holds the array a of size $(k+1,n)$.
<code>x</code>	Holds the vector with the number of elements n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>trans</code>	Must be ' <code>N</code> ', ' <code>C</code> ', or ' <code>T</code> '. The default value is ' <code>N</code> '.
<code>diag</code>	Must be ' <code>N</code> ' or ' <code>U</code> '. The default value is ' <code>N</code> '.

?tpmv

Computes a matrix-vector product using a triangular packed matrix.

Syntax

FORTRAN 77:

```
call stpmv(uplo, trans, diag, n, ap, x, incx)
call dtpmv(uplo, trans, diag, n, ap, x, incx)
call ctpmv(uplo, trans, diag, n, ap, x, incx)
call ztpmv(uplo, trans, diag, n, ap, x, incx)
```

Fortran 95:

```
call tpmv(ap, x [,uplo] [, trans] [,diag])
```

C:

```
void cblas_stpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *ap, float
*x, const MKL_INT incx);
void cblas_dtpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *ap, double
*x, const MKL_INT incx);
```

```
void cblas_ctpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void
*x, const MKL_INT incx);

void cblas_ztpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void
*x, const MKL_INT incx);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?tpmv routines perform one of the matrix-vector operations defined as

$x := A^*x$, or $x := A'^*x$, or $x := \text{conjg}(A')^*x$,

where:

x is an n -element vector,

A is an n -by- n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular: <i>uplo</i> = 'U' or 'u' if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then $x := A^*x$; if <i>trans</i> = 'T' or 't', then $x := A'^*x$; if <i>trans</i> = 'C' or 'c', then $x := \text{conjg}(A')^*x$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix A is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	INTEGER. Specifies the order of the matrix A . The value of <i>n</i> must be at least zero.
<i>ap</i>	REAL for stpmv DOUBLE PRECISION for dtpmv COMPLEX for ctpmv DOUBLE COMPLEX for ztpmv

Array, size at least $((n*(n + 1))/2)$. Before entry with $uplo = 'U'$ or ' u ', the array ap must contain the upper triangular matrix packed sequentially, column-by-column, so that $ap(1)$ contains $a(1,1)$, $ap(2)$ and $ap(3)$ contain $a(1,2)$ and $a(2,2)$ respectively, and so on. Before entry with $uplo = 'L'$ or ' l ', the array ap must contain the lower triangular matrix packed sequentially, column-by-column, so that $ap(1)$ contains $a(1,1)$, $ap(2)$ and $ap(3)$ contain $a(2,1)$ and $a(3,1)$ respectively, and so on. When $diag = 'U'$ or ' u ', the diagonal elements of a are not referenced, but are assumed to be unity.

x	REAL for <code>stpmv</code> DOUBLE PRECISION for <code>dtpmv</code> COMPLEX for <code>ctpmv</code> DOUBLE COMPLEX for <code>ztpmv</code>
	Array, size at least $(1 + (n - 1)*abs(incx))$. Before entry, the incremented array x must contain the n -element vector x .
$incx$	INTEGER. Specifies the increment for the elements of x . The value of $incx$ must not be zero.

Output Parameters

x	Overwritten with the transformed vector x .
-----	---

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tpmv` interface are the following:

ap	Holds the array ap of size $(n*(n+1)/2)$.
x	Holds the vector with the number of elements n .
$uplo$	Must be ' U ' or ' L '. The default value is ' U '.
$trans$	Must be ' N ', ' C ', or ' T '. The default value is ' N '.
$diag$	Must be ' N ' or ' U '. The default value is ' N '.

?tpsv

Solves a system of linear equations whose coefficients are in a triangular packed matrix.

Syntax

FORTRAN 77:

```
call stpsv(uplo, trans, diag, n, ap, x, incx)
call dtpsv(uplo, trans, diag, n, ap, x, incx)
```

```
call ctpsv(uplo, trans, diag, n, ap, x, incx)
call ztpsv(uplo, trans, diag, n, ap, x, incx)
```

Fortran 95:

```
call tpsv(ap, x [,uplo] [, trans] [,diag])
```

C:

```
void cblas_stpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *ap, float
*x, const MKL_INT incx);

void cblas_dtpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *ap, double
*x, const MKL_INT incx);

void cblas_ctpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void
*x, const MKL_INT incx);

void cblas_ztpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void
*x, const MKL_INT incx);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?tpsv routines solve one of the following systems of equations

$A^*x = b$, or $A'^*x = b$, or $\text{conjg}(A')^*x = b$,

where:

b and x are n -element vectors,

A is an n -by- n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

This routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular: <i>uplo</i> = 'U' or 'u' if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the system of equations: if <i>trans</i> = 'N' or 'n', then $A^*x = b$; if <i>trans</i> = 'T' or 't', then $A'^*x = b$;

*if trans= 'C' or 'c', then conjg(A')*x = b.*

diag CHARACTER*1. Specifies whether the matrix *A* is unit triangular:
if diag = 'U' or 'u' then the matrix is unit triangular;
if diag = 'N' or 'n' *diag = CblasNonUnit* , then the matrix is not unit triangular.

n INTEGER. Specifies the order of the matrix *A*. The value of *n* must be at least zero.

ap REAL for stpsv
 DOUBLE PRECISION for dtpsv
 COMPLEX for ctpsv
 DOUBLE COMPLEX for ztpsv
 Array, size at least $((n*(n + 1))/2)$.
 Before entry with *uplo* = 'U' or 'u', the array *ap* must contain the upper triangular part of the triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(1, 2) and *a*(2, 2) respectively, and so on.
 Before entry with *uplo* = 'L' or 'l', the array *ap* must contain the lower triangular part of the triangular matrix packed sequentially, column-by-column, so that *ap*(1) contains *a*(1, 1), *ap*(2) and *ap*(3) contain *a*(2, 1) and *a*(3, 1) respectively, and so on.
 When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced, but are assumed to be unity.

x REAL for stpsv
 DOUBLE PRECISION for dtpsv
 COMPLEX for ctpsv
 DOUBLE COMPLEX for ztpsv
 Array, size at least $(1 + (n - 1)*abs(incx))$. Before entry, the incremented array *x* must contain the *n*-element right-hand side vector *b*.

incx INTEGER. Specifies the increment for the elements of *x*.
 The value of *incx* must not be zero.

Output Parameters

x Overwritten with the solution vector *x*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *tpsv* interface are the following:

ap Holds the array *ap* of size $(n*(n+1)/2)$.

<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

?trmv

Computes a matrix-vector product using a triangular matrix.

Syntax**FORTRAN 77:**

```
call strmv(uplo, trans, diag, n, a, lda, x, incx)
call dtrmv(uplo, trans, diag, n, a, lda, x, incx)
call ctrmv(uplo, trans, diag, n, a, lda, x, incx)
call ztrmv(uplo, trans, diag, n, a, lda, x, incx)
```

Fortran 95:

```
call trmv(a, x [,uplo] [, trans] [,diag])
```

C:

```
void cblas_strmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *a, const
MKL_INT lda, float *x, const MKL_INT incx);

void cblas_dtrmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *a, const
MKL_INT lda, double *x, const MKL_INT incx);

void cblas_ctrmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const
MKL_INT lda, void *x, const MKL_INT incx);

void cblas_ztrmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const
MKL_INT lda, void *x, const MKL_INT incx);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?trmv routines perform one of the following matrix-vector operations defined as

x := *A***x*, or *x* := *A'***x*, or *x* := conjg(*A'*)**x*,

where:

x is an *n*-element vector,

A is an n -by- n unit, or non-unit, upper or lower triangular matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (<code>CblasRowMajor</code>) or column-major (<code>CblasColMajor</code>).
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular: <code>uplo = 'U' or 'u'</code> <code>if uplo = 'L' or 'l', then the matrix is low triangular.</code>
<i>trans</i>	CHARACTER*1. Specifies the operation: <code>if trans= 'N' or 'n', then x := A*x;</code> <code>if trans= 'T' or 't', then x := A'*x;</code> <code>if trans= 'C' or 'c', then x := conjg(A')*x.</code>
<i>diag</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is unit triangular: <code>if diag = 'U' or 'u' then the matrix is unit triangular;</code> <code>if diag = 'N' or 'n' diag = CblasNonUnit , then the matrix is not unit triangular.</code>
<i>n</i>	INTEGER. Specifies the order of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>a</i>	REAL for <code>strmv</code> DOUBLE PRECISION for <code>dtrmv</code> COMPLEX for <code>ctrmv</code> DOUBLE COMPLEX for <code>ztrmv</code> Array, size (lda, n) . Before entry with $\text{uplo} = 'U'$ or $'u'$, the leading n -by- n upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with $\text{uplo} = 'L'$ or $'l'$, the leading n -by- n lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced. When $\text{diag} = 'U'$ or $'u'$, the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	REAL for <code>strmv</code> DOUBLE PRECISION for <code>dtrmv</code> COMPLEX for <code>ctrmv</code> DOUBLE COMPLEX for <code>ztrmv</code> Array, size at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the n -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .

The value of *incx* must not be zero.

Output Parameters

<i>x</i>	Overwritten with the transformed vector <i>x</i> .
----------	--

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trmv` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n,n</i>).
<i>x</i>	Holds the vector with the number of elements <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'.
	The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

?trsv

Solves a system of linear equations whose coefficients are in a triangular matrix.

Syntax

FORTRAN 77:

```
call strsv(uplo, trans, diag, n, a, lda, x, incx)
call dtrsv(uplo, trans, diag, n, a, lda, x, incx)
call ctrsv(uplo, trans, diag, n, a, lda, x, incx)
call ztrsv(uplo, trans, diag, n, a, lda, x, incx)
```

Fortran 95:

```
call trsv(a, x [,uplo] [, trans] [,diag])
```

C:

```
void cblas_strsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *a, const
MKL_INT lda, float *x, const MKL_INT incx);

void cblas_dtrsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *a, const
MKL_INT lda, double *x, const MKL_INT incx);

void cblas_ctrsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const
MKL_INT lda, void *x, const MKL_INT incx);
```

```
void cblas_ztrsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const
MKL_INT lda, void *x, const MKL_INT incx);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?trsv routines solve one of the systems of equations:

$A^*x = b$, or $A'^*x = b$, or $\text{conj}(A')^*x = b$,

where:

b and x are n -element vectors,

A is an n -by- n unit, or non-unit, upper or lower triangular matrix.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (<code>CblasRowMajor</code>) or column-major (<code>CblasColMajor</code>).
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular: <code>uplo = 'U' or 'u'</code> <code>if uplo = 'L' or 'l'</code> , then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the systems of equations: <code>if trans= 'N' or 'n'</code> , then $A^*x = b$; <code>if trans= 'T' or 't'</code> , then $A'^*x = b$; <code>if trans= 'C' or 'c'</code> , then $\text{conj}(A')^*x = b$.
<i>diag</i>	CHARACTER*1. Specifies whether the matrix A is unit triangular: <code>if diag = 'U' or 'u'</code> then the matrix is unit triangular; <code>if diag = 'N' or 'n'</code> , then the matrix is not unit triangular.
<i>n</i>	INTEGER. Specifies the order of the matrix A . The value of n must be at least zero.
<i>a</i>	REAL for strsv DOUBLE PRECISION for dtrsv COMPLEX for ctrsv DOUBLE COMPLEX for ztrsv

Array, size (lda, n) . Before entry with $\text{uplo} = \text{'U'}$ or 'u' , the leading n -by- n upper triangular part of the array a must contain the upper triangular matrix and the strictly lower triangular part of a is not referenced. Before entry with $\text{uplo} = \text{'L'}$ or 'l' , the leading n -by- n lower triangular part of the array a must contain the lower triangular matrix and the strictly upper triangular part of a is not referenced.

When $\text{diag} = \text{'U'}$ or 'u' , the diagonal elements of a are not referenced either, but are assumed to be unity.

lda INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of lda must be at least $\max(1, n)$.

x REAL for strsv

DOUBLE PRECISION for dtrsv

COMPLEX for ctrsv

DOUBLE COMPLEX for ztrsv

Array, size at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array x must contain the n -element right-hand side vector b .

incx INTEGER. Specifies the increment for the elements of x .

The value of incx must not be zero.

Output Parameters

x Overwritten with the solution vector x .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine trsv interface are the following:

a Holds the matrix a of size (n,n) .

x Holds the vector with the number of elements n .

uplo Must be 'U' or 'L' . The default value is 'U' .

trans Must be 'N' , 'C' , or 'T' .

The default value is 'N' .

diag Must be 'N' or 'U' . The default value is 'N' .

BLAS Level 3 Routines

BLAS Level 3 routines perform matrix-matrix operations. [Table "BLAS Level 3 Routine Groups and Their Data Types"](#) lists the BLAS Level 3 routine groups and the data types associated with them.

BLAS Level 3 Routine Groups and Their Data Types

Routine Group	Data Types	Description
?gemm	s, d, c, z	Computes a matrix-matrix product with general matrices.
?hemm	c, z	Computes a matrix-matrix product where one input matrix is Hermitian.
?herk	c, z	Performs a Hermitian rank-k update.
?her2k	c, z	Performs a Hermitian rank-2k update.
?symm	s, d, c, z	Computes a matrix-matrix product where one input matrix is symmetric.
?syrk	s, d, c, z	Performs a symmetric rank-k update.
?syr2k	s, d, c, z	Performs a symmetric rank-2k update.
?trmm	s, d, c, z	Computes a matrix-matrix product where one input matrix is triangular.
?trsm	s, d, c, z	Solves a triangular matrix equation.

Symmetric Multiprocessing Version of Intel® MKL

Many applications spend considerable time executing BLAS routines. This time can be scaled by the number of processors available on the system through using the symmetric multiprocessing (SMP) feature built into the Intel MKL Library. The performance enhancements based on the parallel use of the processors are available without any programming effort on your part.

To enhance performance, the library uses the following methods:

- The BLAS functions are blocked where possible to restructure the code in a way that increases the localization of data reference, enhances cache memory use, and reduces the dependency on the memory bus.
- The code is distributed across the processors to maximize parallelism.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

?gemm

Computes a matrix-matrix product with general matrices.

Syntax**FORTRAN 77:**

```
call sgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call cgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

```
call zgemm(transa, transb, m, k, alpha, a, lda, b, ldb, beta, c, ldc)
call scgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dzgemm(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Fortran 95:

```
call gemm(a, b, c [,transa] [,transb] [,alpha] [,beta])
```

C:

```
void cblas_sgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const float
alpha, const float *a, const MKL_INT lda, const float *b, const MKL_INT ldb, const
float beta, float *c, const MKL_INT ldc);
void cblas_dgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const double
alpha, const double *a, const MKL_INT lda, const double *b, const MKL_INT ldb, const
double beta, double *c, const MKL_INT ldc);
void cblas_cgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void
*i, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void
*i, void *c, const MKL_INT ldc);
void cblas_zgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void
*i, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void
*i, void *c, const MKL_INT ldc);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?gemm routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, with general matrices. The operation is defined as

$$C := \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C,$$

where:

$\text{op}(X)$ is one of $\text{op}(X) = X$, or $\text{op}(X) = X^T$, or $\text{op}(X) = X^H$,

α and β are scalars,

A , B and C are matrices:

$\text{op}(A)$ is an m -by- k matrix,

$\text{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

See also ?gemm3m, BLAS-like extension routines, that use matrix multiplication for similar matrix-matrix operations.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (<code>CblasRowMajor</code>) or column-major (<code>CblasColMajor</code>).
<i>transa</i>	CHARACTER*1. Specifies the form of $\text{op}(A)$ used in the matrix multiplication: <code>if transa = 'N' or 'n', then op(A) = A;</code> <code>if transa = 'T' or 't', then op(A) = A^T;</code> <code>if transa = 'C' or 'c', then op(A) = A^H.</code>
<i>transb</i>	CHARACTER*1. Specifies the form of $\text{op}(B)$ used in the matrix multiplication: <code>if transb = 'N' or 'n', then op(B) = B;</code> <code>if transb = 'T' or 't', then op(B) = B^T;</code> <code>if transb = 'C' or 'c', then op(B) = B^H.</code>
<i>m</i>	INTEGER. Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix C . The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of <i>k</i> must be at least zero.
<i>alpha</i>	REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm, scgomm DOUBLE COMPLEX for zgemm, dzgomm Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for sgemm, scgomm DOUBLE PRECISION for dgemm, dzgomm COMPLEX for cgemm DOUBLE COMPLEX for zgemm Array, size <i>lda</i> by <i>ka</i> , where <i>ka</i> is <i>k</i> when <i>transa</i> = 'N' or 'n', and is <i>m</i> otherwise. Before entry with <i>transa</i> = 'N' or 'n', the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix A , otherwise the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix A .
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>transa</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, m)$, otherwise <i>lda</i> must be at least $\max(1, k)$.

<i>b</i>	REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm, scgemm DOUBLE COMPLEX for zgemm, dzgemm Array, size ldb by kb , where kb is n when $transa = 'N'$ or ' n ', and is k otherwise. Before entry with $transa = 'N'$ or ' n ', the leading k -by- n part of the array <i>b</i> must contain the matrix <i>B</i> , otherwise the leading n -by- k part of the array <i>b</i> must contain the matrix <i>B</i> .
ldb	INTEGER. Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. When $transa = 'N'$ or ' n ', then ldb must be at least $\max(1, k)$, otherwise ldb must be at least $\max(1, n)$.
<i>beta</i>	REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm, scgemm DOUBLE COMPLEX for zgemm, dzgemm Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then <i>c</i> need not be set on input.
<i>c</i>	REAL for sgemm DOUBLE PRECISION for dgemm COMPLEX for cgemm, scgemm DOUBLE COMPLEX for zgemm, dzgemm Array, size ldc by n . Before entry, the leading m -by- n part of the array <i>c</i> must contain the matrix <i>C</i> , except when <i>beta</i> is equal to zero, in which case <i>c</i> need not be set on entry.
ldc	INTEGER. Specifies the leading dimension of <i>c</i> as declared in the calling (sub)program. The value of ldc must be at least $\max(1, m)$.

Output Parameters

c Overwritten by the m -by- n matrix $(alpha * op(A) * op(B) + beta * C)$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gemm` interface are the following:

a Holds the matrix *A* of size (ma, ka) where
 $ka = k$ if $transa = 'N'$,
 $ka = m$ otherwise,

	<i>ma</i> = <i>m</i> if <i>transa</i> = 'N', <i>ma</i> = <i>k</i> otherwise.
<i>b</i>	Holds the matrix <i>B</i> of size (<i>mb,kb</i>) where <i>kb</i> = <i>n</i> if <i>transb</i> = 'N', <i>kb</i> = <i>k</i> otherwise, <i>mb</i> = <i>k</i> if <i>transb</i> = 'N', <i>mb</i> = <i>n</i> otherwise.
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m,n</i>).
<i>transa</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>transb</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?hemm

Computes a matrix-matrix product where one input matrix is Hermitian.

Syntax**FORTRAN 77:**

```
call chemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call zhemm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
```

Fortran 95:

```
call hemm(a, b, c [,side][,uplo] [,alpha][,beta])
```

C:

```
void cblas_chemm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
void cblas_zhemm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?hemm routines compute a scalar-matrix-matrix product using a Hermitian matrix A and a general matrix B and add the result to a scalar-matrix product using a general matrix C . The operation is defined as

$$C := \alpha * A * B + \beta * C$$

or

$$C := \alpha * B * A + \beta * C,$$

where:

α and β are scalars,

A is a Hermitian matrix,

B and C are m -by- n matrices.

Input Parameters

Layout

Specifies whether two-dimensional array storage is row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

side

CHARACTER*1. Specifies whether the Hermitian matrix A appears on the left or right in the operation as follows:

```
if side = 'L' or 'l', then C :=  $\alpha * A * B + \beta * C$ ;
if side = 'R' or 'r', then C :=  $\alpha * B * A + \beta * C$ .
```

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix A is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the Hermitian matrix A is used.

If *uplo* = 'L' or 'l', then the low triangular part of the Hermitian matrix A is used.

m

INTEGER. Specifies the number of rows of the matrix C .

The value of *m* must be at least zero.

n

INTEGER. Specifies the number of columns of the matrix C .

The value of *n* must be at least zero.

alpha

COMPLEX for `chemm`

DOUBLE COMPLEX for `zhemm`

Specifies the scalar α .

a

COMPLEX for `chemm`

DOUBLE COMPLEX for `zhemm`

Array, size (lda, ka) , where ka is m when *side* = 'L' or 'l' and is n otherwise. Before entry with *side* = 'L' or 'l', the m -by- m part of the array *a* must contain the Hermitian matrix, such that when *uplo* = 'U' or 'u', the leading m -by- m upper triangular part of the array *a* must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of *a* is not referenced, and when *uplo* = 'L' or 'l', the

leading m -by- m lower triangular part of the array a must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of a is not referenced.

Before entry with $\text{side} = \text{'R'}$ or 'r' , the n -by- n part of the array a must contain the Hermitian matrix, such that when $\text{uplo} = \text{'U'}$ or 'u' , the leading n -by- n upper triangular part of the array a must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of a is not referenced, and when $\text{uplo} = \text{'L'}$ or 'l' , the leading n -by- n lower triangular part of the array a must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of a is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

lda	INTEGER. Specifies the leading dimension of a as declared in the calling (sub) program. When $\text{side} = \text{'L'}$ or 'l' then lda must be at least $\max(1, m)$, otherwise lda must be at least $\max(1, n)$.
b	<p>COMPLEX for <code>chemm</code></p> <p>DOUBLE COMPLEX for <code>zhemm</code></p> <p>Array, size ldb by n.</p> <p>Before entry, the leading m-by-n part of the array b must contain the matrix B.</p>
ldb	INTEGER. Specifies the leading dimension of b as declared in the calling (sub)program. The value of ldb must be at least $\max(1, m)$.
β	<p>COMPLEX for <code>chemm</code></p> <p>DOUBLE COMPLEX for <code>zhemm</code></p> <p>Specifies the scalar β.</p> <p>When β is supplied as zero, then c need not be set on input.</p>
c	<p>COMPLEX for <code>chemm</code></p> <p>DOUBLE COMPLEX for <code>zhemm</code></p> <p>Array, size (c, n). Before entry, the leading m-by-n part of the array c must contain the matrix C, except when β is zero, in which case c need not be set on entry.</p>
ldc	INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program. The value of ldc must be at least $\max(1, m)$.

Output Parameters

c Overwritten by the m -by- n updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hemm` interface are the following:

<i>a</i>	Holds the matrix A of size (k,k) where $k = m$ if $\text{side} = \text{'L'}$, $k = n$ otherwise.
<i>b</i>	Holds the matrix B of size (m,n) .
<i>c</i>	Holds the matrix C of size (m,n) .
<i>side</i>	Must be 'L' or 'R' . The default value is 'L' .
<i>uplo</i>	Must be 'U' or 'L' . The default value is 'U' .
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?herk*Performs a Hermitian rank-k update.***Syntax****FORTRAN 77:**

```
call cherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call zherk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
```

Fortran 95:

```
call herk(a, c [,uplo] [, trans] [,alpha][,beta])
```

C:

```
void cblas_cherk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const float alpha, const void
*a, const MKL_INT lda, const float beta, void *c, const MKL_INT ldc);
void cblas_zherk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const double alpha, const void
*a, const MKL_INT lda, const double beta, void *c, const MKL_INT ldc);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?herk routines perform a rank-k matrix-matrix operation using a general matrix A and a Hermitian matrix C . The operation is defined as

```
C := alpha*A*AH + beta*C,
```

or

```
C := alpha*AH*A + beta*C,
```

where:

alpha and *beta* are real scalars,

C is an n -by- n Hermitian matrix,

A is an n -by- k matrix in the first case and a k -by- n matrix in the second case.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (<code>CblasRowMajor</code>) or column-major (<code>CblasColMajor</code>).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>c</i> is used.
<i>trans</i>	CHARACTER*1. Specifies the operation: <code>if trans= 'N' or 'n', then C := alpha*A*A^H + beta*C;</code> <code>if trans= 'C' or 'c', then C := alpha*A^H*A + beta*C.</code>
<i>n</i>	INTEGER. Specifies the order of the matrix <i>C</i> . The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. With <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the matrix <i>A</i> , and with <i>trans</i> = 'C' or 'c', <i>k</i> specifies the number of rows of the matrix <i>A</i> . The value of <i>k</i> must be at least zero.
<i>alpha</i>	REAL for <code>cherk</code> DOUBLE PRECISION for <code>zherk</code> Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for <code>cherk</code> DOUBLE COMPLEX for <code>zherk</code> Array, size (<i>lda</i> , <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>a</i> , otherwise the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least <code>max(1, n)</code> , otherwise <i>lda</i> must be at least <code>max(1, k)</code> .
<i>beta</i>	REAL for <code>cherk</code> DOUBLE PRECISION for <code>zherk</code> Specifies the scalar <i>beta</i> .
<i>c</i>	COMPLEX for <code>cherk</code> DOUBLE COMPLEX for <code>zherk</code> Array, size <i>ldc</i> by <i>n</i> .

Before entry with `uplo = 'U'` or `'u'`, the leading n -by- n upper triangular part of the array c must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of c is not referenced.

Before entry with `uplo = 'L'` or `'l'`, the leading n -by- n lower triangular part of the array c must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of c is not referenced.

The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

`ldc`

INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program. The value of `ldc` must be at least $\max(1, n)$.

Output Parameters

`c`

With `uplo = 'U'` or `'u'`, the upper triangular part of the array c is overwritten by the upper triangular part of the updated matrix.

With `uplo = 'L'` or `'l'`, the lower triangular part of the array c is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `herk` interface are the following:

`a`

Holds the matrix A of size (ma, ka) where
 $ka = k$ if `transa= 'N'`,
 $ka = n$ otherwise,
 $ma = n$ if `transa= 'N'`,
 $ma = k$ otherwise.

`c`

Holds the matrix C of size (n,n) .

`uplo`

Must be `'U'` or `'L'`. The default value is `'U'`.

`trans`

Must be `'N'` or `'C'`. The default value is `'N'`.

`alpha`

The default value is 1.

`beta`

The default value is 0.

?her2k

Performs a Hermitian rank- $2k$ update.

Syntax

FORTRAN 77:

```
call cher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zher2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Fortran 95:

```
call her2k(a, b, c [,uplo][,trans] [,alpha][,beta])
```

C:

```
void cblas_cher2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *b, const MKL_INT ldb, const float beta, void *c,
const MKL_INT ldc);

void cblas_zher2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *b, const MKL_INT ldb, const double beta, void *c,
const MKL_INT ldc);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?her2k routines perform a rank-2k matrix-matrix operation using general matrices A and B and a Hermitian matrix C . The operation is defined as

$$C := \alpha A B^H + \text{conj}(\alpha) B A^H + \beta C,$$

or

$$C := \alpha B^H A + \text{conj}(\alpha) A^H B + \beta C,$$

where:

α is a scalar and β is a real scalar,

C is an n -by- n Hermitian matrix,

A and B are n -by- k matrices in the first case and k -by- n matrices in the second case.

Input Parameters

Layout

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the array c is used.

If $uplo = 'U'$ or ' u ', then the upper triangular of the array c is used.

If $uplo = 'L'$ or ' l ', then the low triangular of the array c is used.

trans

CHARACTER*1. Specifies the operation:

If $trans= 'N'$ or ' n ', then $C:=\alpha A B^H + \alpha B A^H + \beta C;$

If $trans= 'C'$ or ' c ', then $C:=\alpha A^H B + \alpha B^H A + \beta C.$

n

INTEGER. Specifies the order of the matrix C . The value of n must be at least zero.

<i>k</i>	INTEGER. With <i>trans</i> = 'N' or 'n' specifies the number of columns of the matrix <i>A</i> , and with <i>trans</i> = 'C' or 'c', <i>k</i> specifies the number of rows of the matrix <i>A</i> .
	The value of <i>k</i> must be at least equal to zero.
<i>alpha</i>	COMPLEX for <i>cher2k</i> DOUBLE COMPLEX for <i>zher2k</i> Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for <i>cher2k</i> DOUBLE COMPLEX for <i>zher2k</i> Array, size (<i>lda</i> , <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> , otherwise the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n' <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.
<i>beta</i>	REAL for <i>cher2k</i> DOUBLE PRECISION for <i>zher2k</i> Specifies the scalar <i>beta</i> .
<i>b</i>	COMPLEX for <i>cher2k</i> DOUBLE COMPLEX for <i>zher2k</i> Array, size (<i>ldb</i> , <i>kb</i>), where <i>kb</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i> -by- <i>k</i> part of the array <i>b</i> must contain the matrix <i>B</i> , otherwise the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>ldb</i>	INTEGER. Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>ldb</i> must be at least $\max(1, n)$, otherwise <i>ldb</i> must be at least $\max(1, k)$.
<i>c</i>	COMPLEX for <i>cher2k</i> DOUBLE COMPLEX for <i>zher2k</i> Array, size <i>ldc</i> by <i>n</i> . Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>c</i> must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of <i>c</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>c</i> must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of <i>c</i> is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least $\max(1, n)$.

Output Parameters

c With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `her2k` interface are the following:

a Holds the matrix *A* of size (*ma,ka*) where

```
ka = k if trans = 'N',
ka = n otherwise,
ma = n if trans = 'N',
ma = k otherwise.
```

b Holds the matrix *B* of size (*mb,kb*) where

```
kb = k if trans = 'N',
kb = n otherwise,
mb = n if trans = 'N',
mb = k otherwise.
```

c Holds the matrix *C* of size (*n,n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

trans Must be 'N' or 'C'. The default value is 'N'.

alpha The default value is 1.

beta The default value is 0.

?symm

Computes a matrix-matrix product where one input matrix is symmetric.

Syntax

FORTRAN 77:

```
call ssymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call dsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
```

```
call csymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
call zsymm(side, uplo, m, n, alpha, a, lda, b, ldb, beta, c, ldc)
```

Fortran 95:

```
call symm(a, b, c [,side] [,uplo] [,alpha] [,beta])
```

C:

```
void cblas_ssimm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const MKL_INT m, const MKL_INT n, const float alpha, const float *a, const
MKL_INT lda, const float *b, const MKL_INT ldb, const float beta, float *c, const
MKL_INT ldc);

void cblas_dsymm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const MKL_INT m, const MKL_INT n, const double alpha, const double *a, const
MKL_INT lda, const double *b, const MKL_INT ldb, const double beta, double *c, const
MKL_INT ldc);

void cblas_csymm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const
MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const
MKL_INT ldc);

void cblas_zsymm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const
MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const
MKL_INT ldc);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?symm routines compute a scalar-matrix-matrix product with one symmetric matrix and add the result to a scalar-matrix product. The operation is defined as

```
C := alpha*A*B + beta*C,
```

or

```
C := alpha*B*A + beta*C,
```

where:

alpha and *beta* are scalars,

A is a symmetric matrix,

B and *C* are *m*-by-*n* matrices.

Input Parameters

Layout

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

side

CHARACTER*1. Specifies whether the symmetric matrix *A* appears on the left or right in the operation:

```
if side = 'L' or 'l', then C := alpha*A*B + beta*C;
```

`if side = 'R' or 'r', then C := alpha*B*A + beta*C.`

`uplo`

CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is used:

`if uplo = 'U' or 'u', then the upper triangular part is used;`

`if uplo = 'L' or 'l', then the lower triangular part is used.`

`m`

INTEGER. Specifies the number of rows of the matrix C .

The value of m must be at least zero.

`n`

INTEGER. Specifies the number of columns of the matrix C .

The value of n must be at least zero.

`alpha`

REAL for ssymm

DOUBLE PRECISION for dsymm

COMPLEX for csymm

DOUBLE COMPLEX for zsymm

Specifies the scalar α .

`a`

REAL for ssymm

DOUBLE PRECISION for dsymm

COMPLEX for csymm

DOUBLE COMPLEX for zsymm

Array, size (lda, ka), where ka is m when $side = 'L'$ or ' l ' and is n otherwise.

Before entry with $side = 'L'$ or ' l ', the m -by- m part of the array a must contain the symmetric matrix, such that when $uplo = 'U'$ or ' u ', the leading m -by- m upper triangular part of the array a must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of a is not referenced, and when $side = 'L'$ or ' l ', the leading m -by- m lower triangular part of the array a must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of a is not referenced.

Before entry with $side = 'R'$ or ' r ', the n -by- n part of the array a must contain the symmetric matrix, such that when $uplo = 'U'$ or ' u ' the array a must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of a is not referenced, and when $side = 'L'$ or ' l ', the leading n -by- n lower triangular part of the array a must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of a is not referenced.

`lda`

INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. When $side = 'L'$ or ' l ' then lda must be at least $\max(1, m)$, otherwise lda must be at least $\max(1, n)$.

`b`

REAL for ssymm

DOUBLE PRECISION for dsymm

COMPLEX for `csymm`

DOUBLE COMPLEX for `zsymm`

Array, size (l_{db}, n) . Before entry, the leading m -by- n part of the array b must contain the matrix B .

ldb INTEGER. Specifies the leading dimension of b as declared in the calling (sub)program. The value of ldb must be at least $\max(1, m)$.

beta REAL for `ssymm`

DOUBLE PRECISION for `dsymm`

COMPLEX for `csymm`

DOUBLE COMPLEX for `zsymm`

Specifies the scalar β .

When β is set to zero, then c need not be set on input.

c REAL for `ssymm`

DOUBLE PRECISION for `dsymm`

COMPLEX for `csymm`

DOUBLE COMPLEX for `zsymm`

Array, size l_{dc} by n . Before entry, the leading m -by- n part of the array c must contain the matrix C , except when β is zero, in which case c need not be set on entry.

ldc INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program. The value of ldc must be at least $\max(1, m)$.

Output Parameters

c Overwritten by the m -by- n updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `symm` interface are the following:

a Holds the matrix A of size (k,k) where

$k = m$ if `side = 'L'`,

$k = n$ otherwise.

b Holds the matrix B of size (m,n) .

c Holds the matrix C of size (m,n) .

side Must be '`L`' or '`R`'. The default value is '`L`'.

uplo Must be '`U`' or '`L`'. The default value is '`U`'.

<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?syrk*Performs a symmetric rank-k update.***Syntax****FORTRAN 77:**

```
call ssyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call dsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call csyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
call zsyrk(uplo, trans, n, k, alpha, a, lda, beta, c, ldc)
```

Fortran 95:

```
call syrk(a, c [,uplo] [, trans] [,alpha][,beta])
```

C:

```
void cblas_ssyrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const float alpha, const float
*a, const MKL_INT lda, const float beta, float *c, const MKL_INT ldc);
void cblas_dsyrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const double alpha, const
double *a, const MKL_INT lda, const double beta, double *c, const MKL_INT ldc);
void cblas_csrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *beta, void *c, const MKL_INT ldc);
void cblas_zsyrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *beta, void *c, const MKL_INT ldc);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?syrk routines perform a rank-k matrix-matrix operation for a symmetric matrix *C* using a general matrix *A*. The operation is defined as

```
C := alpha*A*A' + beta*C,
```

or

```
C := alpha*A'*A + beta*C,
```

where:

alpha and *beta* are scalars,

C is an *n*-by-*n* symmetric matrix,

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>c</i> is used.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then <i>C</i> := <i>alpha</i> * <i>A</i> * <i>A</i> ' + <i>beta</i> * <i>C</i> ; if <i>trans</i> = 'T' or 't', then <i>C</i> := <i>alpha</i> * <i>A</i> '* <i>A</i> + <i>beta</i> * <i>C</i> ; if <i>trans</i> = 'C' or 'c', then <i>C</i> := <i>alpha</i> * <i>A</i> '* <i>A</i> + <i>beta</i> * <i>C</i> .
<i>n</i>	INTEGER. Specifies the order of the matrix <i>C</i> . The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the matrix <i>a</i> , and on entry with <i>trans</i> = 'T', 't', 'C', or 'c', <i>k</i> specifies the number of rows of the matrix <i>a</i> . The value of <i>k</i> must be at least zero.
<i>alpha</i>	REAL for ssyrk DOUBLE PRECISION for dsyrk COMPLEX for csyrk DOUBLE COMPLEX for zsyrk Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for ssyrk DOUBLE PRECISION for dsyrk COMPLEX for csyrk DOUBLE COMPLEX for zsyrk Array, size (<i>lda</i> , <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> , otherwise the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.
<i>beta</i>	REAL for ssyrk DOUBLE PRECISION for dsyrk COMPLEX for csyrk DOUBLE COMPLEX for zsyrk

Specifies the scalar *beta*.

c REAL for ssyrk

DOUBLE PRECISION for dsyrk

COMPLEX for csyrk

DOUBLE COMPLEX for zsydk

Array, size (*ldc*, *n*). Before entry with *uplo* = 'U' or 'u', the leading *n*-by-*n* upper triangular part of the array *c* must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of *c* is not referenced.

Before entry with *uplo* = 'L' or 'l', the leading *n*-by-*n* lower triangular part of the array *c* must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of *c* is not referenced.

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program. The value of *ldc* must be at least $\max(1, n)$.

Output Parameters

c With *uplo* = 'U' or 'u', the upper triangular part of the array *c* is overwritten by the upper triangular part of the updated matrix.

With *uplo* = 'L' or 'l', the lower triangular part of the array *c* is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *sydk* interface are the following:

a Holds the matrix *A* of size (*ma*, *ka*) where

ka = *k* if *transa*= 'N',

ka = *n* otherwise,

ma = *n* if *transa*= 'N',

ma = *k* otherwise.

c Holds the matrix *C* of size (*n*, *n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

trans Must be 'N', 'C', or 'T'.

The default value is 'N'.

alpha The default value is 1.

beta The default value is 0.

?syr2k

Performs a symmetric rank-2k update.

Syntax

FORTRAN 77:

```
call ssyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call dsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call csyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zsyr2k(uplo, trans, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Fortran 95:

```
call syr2k(a, b, c [,uplo][,trans] [,alpha][,beta])
```

C:

```
void cblas_ssyr2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const float alpha, const float
*a, const MKL_INT lda, const float *b, const MKL_INT ldb, const float beta, float *c,
const MKL_INT ldc);
void cblas_dsy2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const double alpha, const
double *a, const MKL_INT lda, const double *b, const MKL_INT ldb, const double beta,
double *c, const MKL_INT ldc);
void cblas_csy2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c,
const MKL_INT ldc);
void cblas_zsy2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c,
const MKL_INT ldc);
```

Include Files

- **Fortran:** mkl.fi
- **Fortran 95:** blas.f90
- **C:** mkl.h

Description

The ?syr2k routines perform a rank-2k matrix-matrix operation for a symmetric matrix C using general matrices A and B . The operation is defined as

```
C := alpha*A*B' + alpha*B*A' + beta*C,
```

or

```
C := alpha*A'*B + alpha*B'*A + beta*C,
```

where:

α and β are scalars,

C is an n -by- n symmetric matrix,

A and B are n -by- k matrices in the first case, and k -by- n matrices in the second case.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>c</i> is used.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then <i>C</i> := <i>alpha</i> * <i>A</i> * <i>B</i> ' + <i>alpha</i> * <i>B</i> * <i>A</i> ' + <i>beta</i> * <i>C</i> ; if <i>trans</i> = 'T' or 't', then <i>C</i> := <i>alpha</i> * <i>A</i> '* <i>B</i> + <i>alpha</i> * <i>B</i> '* <i>A</i> + <i>beta</i> * <i>C</i> ; if <i>trans</i> = 'C' or 'c', then <i>C</i> := <i>alpha</i> * <i>A</i> '* <i>B</i> + <i>alpha</i> * <i>B</i> '* <i>A</i> + <i>beta</i> * <i>C</i> .
<i>n</i>	INTEGER. Specifies the order of the matrix <i>C</i> . The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the matrices <i>A</i> and <i>B</i> , and on entry with <i>trans</i> = 'T' or 't' or 'C' or 'c', <i>k</i> specifies the number of rows of the matrices <i>A</i> and <i>B</i> . The value of <i>k</i> must be at least zero.
<i>alpha</i>	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyrr2k Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyrr2k Array, size (<i>lda</i> , <i>ka</i>), where <i>ka</i> is <i>k</i> when <i>trans</i> = 'N' or 'n', and is <i>n</i> otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading <i>n</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> , otherwise the leading <i>k</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.
<i>b</i>	REAL for ssyr2k DOUBLE PRECISION for dsyr2k COMPLEX for csyr2k DOUBLE COMPLEX for zsyrr2k

Array, size ldb by kb where kb is k when $trans = 'N'$ or ' n ' and is n otherwise. Before entry with $trans = 'N'$ or ' n ', the leading n -by- k part of the array b must contain the matrix B , otherwise the leading k -by- n part of the array b must contain the matrix B .

ldb INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. When $trans = 'N'$ or ' n ', then ldb must be at least $\max(1, n)$, otherwise ldb must be at least $\max(1, k)$.

beta REAL for ssyr2k
DOUBLE PRECISION for dsyr2k
COMPLEX for csyr2k
DOUBLE COMPLEX for zsyrr2k
Specifies the scalar *beta*.

c REAL for ssyr2k
DOUBLE PRECISION for dsyr2k
COMPLEX for csyr2k
DOUBLE COMPLEX for zsyrr2k

Array, size (ldc, n) . Before entry with $uplo = 'U'$ or ' u ', the leading n -by- n upper triangular part of the array c must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of c is not referenced.

Before entry with $uplo = 'L'$ or ' l ', the leading n -by- n lower triangular part of the array c must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of c is not referenced.

ldc INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program. The value of ldc must be at least $\max(1, n)$.

Output Parameters

c With $uplo = 'U'$ or ' u ', the upper triangular part of the array c is overwritten by the upper triangular part of the updated matrix.
With $uplo = 'L'$ or ' l ', the lower triangular part of the array c is overwritten by the lower triangular part of the updated matrix.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `syr2k` interface are the following:

a Holds the matrix A of size (ma, ka) where
 $ka = k$ if $trans = 'N'$,
 $ka = n$ otherwise,
 $ma = n$ if $trans = 'N'$,

	$ma = k$ otherwise.
<i>b</i>	Holds the matrix B of size (mb, kb) where $kb = k$ if $trans = 'N'$, $kb = n$ otherwise, $mb = n$ if $trans = 'N'$, $mb = k$ otherwise.
<i>c</i>	Holds the matrix C of size (n, n).
<i>uplo</i>	Must be ' U ' or ' L '. The default value is ' U '.
<i>trans</i>	Must be ' N ', ' C ', or ' T '. The default value is ' N '.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?trmm

Computes a matrix-matrix product where one input matrix is triangular.

Syntax**FORTRAN 77:**

```
call strmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call dtrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ctrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ztrmm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
```

Fortran 95:

```
call trmm(a, b [,side] [, uplo] [,transa] [,diag] [,alpha])
```

C:

```
void cblas_strmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const float alpha, const float *a, const MKL_INT lda, float *b, const MKL_INT ldb);

void cblas_dtrmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const double alpha, const double *a, const MKL_INT lda, double *b, const MKL_INT ldb);

void cblas_ctrmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT ldb);
```

```
void cblas_ztrmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT ldb);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?trmm routines compute a scalar-matrix-matrix product with one triangular matrix. The operation is defined as

```
B := alpha*op(A)*B
```

or

```
B := alpha*B*op(A)
```

where:

alpha is a scalar,

B is an *m*-by-*n* matrix,

A is a unit, or non-unit, upper or lower triangular matrix

op(A) is one of *op(A) = A*, or *op(A) = A'*, or *op(A) = conjg(A')*.

Input Parameters

Layout

Specifies whether two-dimensional array storage is row-major (*CblasRowMajor*) or column-major (*CblasColMajor*).

side

CHARACTER*1. Specifies whether *op(A)* appears on the left or right of *B* in the operation:

```
if side = 'L' or 'l', then B := alpha*op(A)*B;
if side = 'R' or 'r', then B := alpha*B*op(A).
```

uplo

CHARACTER*1. Specifies whether the matrix *A* is upper or lower triangular:

```
uplo = 'U' or 'u'
if uplo = 'L' or 'l', then the matrix is low triangular.
```

transa

CHARACTER*1. Specifies the form of *op(A)* used in the matrix multiplication:

```
if transa= 'N' or 'n', then op(A) = A;
if transa= 'T' or 't', then op(A) = A';
if transa= 'C' or 'c', then op(A) = conjg(A').
```

diag

CHARACTER*1. Specifies whether the matrix *A* is unit triangular:

```
if diag = 'U' or 'u' then the matrix is unit triangular;
if diag = 'N' or 'n' diag = CblasNonUnit , then the matrix is not unit
triangular.
```

<i>m</i>	INTEGER. Specifies the number of rows of <i>B</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of <i>B</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	<p>REAL for strmm</p> <p>DOUBLE PRECISION for dtrmm</p> <p>COMPLEX for ctrmm</p> <p>DOUBLE COMPLEX for ztrmm</p> <p>Specifies the scalar <i>alpha</i>.</p> <p>When <i>alpha</i> is zero, then <i>a</i> is not referenced and <i>b</i> need not be set before entry.</p>
<i>a</i>	<p>REAL for strmm</p> <p>DOUBLE PRECISION for dtrmm</p> <p>COMPLEX for ctrmm</p> <p>DOUBLE COMPLEX for ztrmm</p> <p>Array, size <i>lda</i> by <i>k</i>, where <i>k</i> is <i>m</i> when <i>side</i> = 'L' or 'l' and is <i>n</i> when <i>side</i> = 'R' or 'r'. Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>k</i> by <i>k</i> upper triangular part of the array <i>a</i> must contain the upper triangular matrix and the strictly lower triangular part of <i>a</i> is not referenced.</p> <p>Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>k</i> by <i>k</i> lower triangular part of the array <i>a</i> must contain the lower triangular matrix and the strictly upper triangular part of <i>a</i> is not referenced.</p> <p>When <i>diag</i> = 'U' or 'u', the diagonal elements of <i>a</i> are not referenced either, but are assumed to be unity.</p>
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>side</i> = 'L' or 'l', then <i>lda</i> must be at least $\max(1, m)$, when <i>side</i> = 'R' or 'r', then <i>lda</i> must be at least $\max(1, n)$.
<i>b</i>	<p>REAL for strmm</p> <p>DOUBLE PRECISION for dtrmm</p> <p>COMPLEX for ctrmm</p> <p>DOUBLE COMPLEX for ztrmm</p> <p>Array, size <i>ldb</i> by <i>n</i>.</p> <p>Before entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i>.</p>
<i>ldb</i>	INTEGER. Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. The value of <i>ldb</i> must be at least $\max(1, m)$.

Output Parameters

<i>b</i>	Overwritten by the transformed matrix.
----------	--

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trmm` interface are the following:

<i>a</i>	Holds the matrix A of size (k,k) where $k = m$ if <i>side</i> = 'L', $k = n$ otherwise.
<i>b</i>	Holds the matrix B of size (m,n) .
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>transa</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.
<i>alpha</i>	The default value is 1.

?trsm

Solves a triangular matrix equation.

Syntax

FORTRAN 77:

```
call strsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call dtrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ctrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
call ztrsm(side, uplo, transa, diag, m, n, alpha, a, lda, b, ldb)
```

Fortran 95:

```
call trsm(a, b [,side] [, uplo] [,transa][,diag] [,alpha])
```

C:

```
void cblas_strsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const
MKL_INT n, const float alpha, const float *a, const MKL_INT lda, float *b, const
MKL_INT ldb);

void cblas_dtrsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const
MKL_INT n, const double alpha, const double *a, const MKL_INT lda, double *b, const
MKL_INT ldb);

void cblas_ctrsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO
uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const
MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT
ldb);
```

```
void cblas_ztrsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT ldb);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?trsm routines solve one of the following matrix equations:

```
op(A)*X = alpha*B,
```

or

```
X*op(A) = alpha*B,
```

where:

alpha is a scalar,

X and *B* are *m*-by-*n* matrices,

A is a unit, or non-unit, upper or lower triangular matrix

op(A) is one of *op(A)* = *A*, or *op(A)* = *A'*, or *op(A)* = *conjg(A')*.

The matrix *B* is overwritten by the solution matrix *X*.

Input Parameters

Layout

Specifies whether two-dimensional array storage is row-major (*CblasRowMajor*) or column-major (*CblasColMajor*).

side

CHARACTER*1. Specifies whether *op(A)* appears on the left or right of *X* in the equation:

```
if side = 'L' or 'l', then op(A)*X = alpha*B;
if side = 'R' or 'r', then X*op(A) = alpha*B.
```

uplo

CHARACTER*1. Specifies whether the matrix *A* is upper or lower triangular:

```
uplo = 'U' or 'u'
if uplo = 'L' or 'l', then the matrix is low triangular.
```

transa

CHARACTER*1. Specifies the form of *op(A)* used in the matrix multiplication:

```
if transa= 'N' or 'n', then op(A) = A;
if transa= 'T' or 't';
if transa= 'C' or 'c', then op(A) = conjg(A').
```

diag

CHARACTER*1. Specifies whether the matrix *A* is unit triangular:

```
if diag = 'U' or 'u' then the matrix is unit triangular;
```

if diag = 'N' or 'n' *diag = CblasNonUnit* , then the matrix is not unit triangular.

m INTEGER. Specifies the number of rows of *B*. The value of *m* must be at least zero.

n INTEGER. Specifies the number of columns of *B*. The value of *n* must be at least zero.

alpha REAL for strsm

DOUBLE PRECISION for dtrsm

COMPLEX for ctrsm

DOUBLE COMPLEX for ztrsm

Specifies the scalar *alpha*.

When *alpha* is zero, then *a* is not referenced and *b* need not be set before entry.

a REAL for strsm

DOUBLE PRECISION for dtrsm

COMPLEX for ctrsm

DOUBLE COMPLEX for ztrsm

Array, size (*lda*, *k*) , where *k* is *m* when *side* = 'L' or 'l' and is *n* when *side* = 'R' or 'r'. Before entry with *uplo* = 'U' or 'u', the leading *k* by *k* upper triangular part of the array *a* must contain the upper triangular matrix and the strictly lower triangular part of *a* is not referenced.

Before entry with *uplo* = 'L' or 'l' lower triangular part of the array *a* must contain the lower triangular matrix and the strictly upper triangular part of *a* is not referenced.

When *diag* = 'U' or 'u', the diagonal elements of *a* are not referenced either, but are assumed to be unity.

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. When *side* = 'L' or 'l', then *lda* must be at least $\max(1, m)$, when *side* = 'R' or 'r', then *lda* must be at least $\max(1, n)$.

b REAL for strsm

DOUBLE PRECISION for dtrsm

COMPLEX for ctrsm

DOUBLE COMPLEX for ztrsm

Array, size (*ldb*, *n*) . Before entry, the leading *m*-by-*n* part of the array *b* must contain the right-hand side matrix *B*.

ldb INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program. The value of *ldb* must be at least $\max(1, +m)$.

Output Parameters

b Overwritten by the solution matrix X .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trsm` interface are the following:

<i>a</i>	Holds the matrix A of size (k,k) where $k = m$ if <code>side = 'L'</code> , $k = n$ otherwise.
<i>b</i>	Holds the matrix B of size (m,n) .
<i>side</i>	Must be ' <code>L</code> ' or ' <code>R</code> '. The default value is ' <code>L</code> '.
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<i>transa</i>	Must be ' <code>N</code> ', ' <code>C</code> ', or ' <code>T</code> '.
	The default value is ' <code>N</code> '.
<i>diag</i>	Must be ' <code>N</code> ' or ' <code>U</code> '. The default value is ' <code>N</code> '.
<i>alpha</i>	The default value is 1.

Sparse BLAS Level 1 Routines

This section describes Sparse BLAS Level 1, an extension of BLAS Level 1 included in the Intel® Math Kernel Library beginning with the Intel MKL release 2.1. Sparse BLAS Level 1 is a group of routines and functions that perform a number of common vector operations on sparse vectors stored in compressed form.

Sparse vectors are those in which the majority of elements are zeros. Sparse BLAS routines and functions are specially implemented to take advantage of vector sparsity. This allows you to achieve large savings in computer time and memory. If nz is the number of non-zero vector elements, the computer time taken by Sparse BLAS operations will be $O(nz)$.

Vector Arguments

Compressed sparse vectors. Let a be a vector stored in an array, and assume that the only non-zero elements of a are the following:

```
a(k1), a(k2), a(k3) . . . a(knz),
```

where nz is the total number of non-zero elements in a .

In Sparse BLAS, this vector can be represented in compressed form by two FORTRAN arrays, x (values) and $indx$ (indices). Each array has nz elements:

```
x(1)=a(k1), x(2)=a(k2), . . . x(nz)= a(knz),
```

```
indx(1)=k1, indx(2)=k2, . . . indx(nz)= knz.
```

Thus, a sparse vector is fully determined by the triple $(nz, x, indx)$. If you pass a negative or zero value of nz to Sparse BLAS, the subroutines do not modify any arrays or variables.

Full-storage vectors. Sparse BLAS routines can also use a vector argument fully stored in a single FORTRAN array (a full-storage vector). If y is a full-storage vector, its elements must be stored contiguously: the first element in $y(1)$, the second in $y(2)$, and so on. This corresponds to an increment $incy = 1$ in BLAS Level 1. No increment value for full-storage vectors is passed as an argument to Sparse BLAS routines or functions.

Naming Conventions

Similar to BLAS, the names of Sparse BLAS subprograms have prefixes that determine the data type involved: s and d for single- and double-precision real; c and z for single- and double-precision complex respectively.

If a Sparse BLAS routine is an extension of a "dense" one, the subprogram name is formed by appending the suffix i (standing for *indexed*) to the name of the corresponding "dense" subprogram. For example, the Sparse BLAS routine `saxpyi` corresponds to the BLAS routine `saxpy`, and the Sparse BLAS function `cdotci` corresponds to the BLAS function `cdotc`.

Routines and Data Types

Routines and data types supported in the Intel MKL implementation of Sparse BLAS are listed in [Table "Sparse BLAS Routines and Their Data Types"](#).

Sparse BLAS Routines and Their Data Types

Routine/ Function	Data Types	Description
?axpyi	s, d, c, z	Scalar-vector product plus vector (routines)
?doti	s, d	Dot product (functions)
?dotci	c, z	Complex dot product conjugated (functions)
?dotui	c, z	Complex dot product unconjugated (functions)
?gthr	s, d, c, z	Gathering a full-storage sparse vector into compressed form $nz, x, indx$ (routines)
?gthrz	s, d, c, z	Gathering a full-storage sparse vector into compressed form and assigning zeros to gathered elements in the full-storage vector (routines)
?roti	s, d	Givens rotation (routines)
?sctr	s, d, c, z	Scattering a vector from compressed form to full-storage form (routines)

BLAS Level 1 Routines That Can Work With Sparse Vectors

The following BLAS Level 1 routines will give correct results when you pass to them a compressed-form array x (with the increment $incx=1$):

?asum	sum of absolute values of vector elements
?copy	copying a vector
?nrm2	Euclidean norm of a vector
?scal	scaling a vector

`i?amax` index of the element with the largest absolute value for real flavors, or the largest sum $|\operatorname{Re}(x(i))| + |\operatorname{Im}(x(i))|$ for complex flavors.

`i?amin` index of the element with the smallest absolute value for real flavors, or the smallest sum $|\operatorname{Re}(x(i))| + |\operatorname{Im}(x(i))|$ for complex flavors.

The result i returned by `i?amax` and `i?amin` should be interpreted as index in the compressed-form array, so that the largest (smallest) value is $x(i)$; the corresponding index in full-storage array is $\operatorname{indx}(i)$.

You can also call `?rotg` to compute the parameters of Givens rotation and then pass these parameters to the Sparse BLAS routines `?roti`.

?axpyi

Adds a scalar multiple of compressed sparse vector to a full-storage vector.

Syntax

FORTRAN 77:

```
call saxpyi(nz, a, x, indx, y)
call daxpyi(nz, a, x, indx, y)
call caxpyi(nz, a, x, indx, y)
call zaxpyi(nz, a, x, indx, y)
```

Fortran 95:

```
call axpyi(x, indx, y [, a])
```

C:

```
void cblas_saxpyi (const MKL_INT nz, const float a, const float *x, const MKL_INT
*indx, float *y);
void cblas_daxpyi (const MKL_INT nz, const double a, const double *x, const MKL_INT
*indx, double *y);
void cblas_caxpyi (const MKL_INT nz, const void *a, const void *x, const MKL_INT *indx,
void *y);
void cblas_zaxpyi (const MKL_INT nz, const void *a, const void *x, const MKL_INT *indx,
void *y);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The `?axpyi` routines perform a vector-vector operation defined as

```
y := a*x + y
```

where:

a is a scalar,

x is a sparse vector stored in compressed form,

y is a vector in full storage form.

The `?axpyi` routines reference or modify only the elements of y whose indices are listed in the array $indx$.

The values in $indx$ must be distinct.

Input Parameters

nz	INTEGER. The number of elements in x and $indx$.
a	REAL for <code>saxpyi</code> DOUBLE PRECISION for <code>daxpyi</code> COMPLEX for <code>caxpyi</code> DOUBLE COMPLEX for <code>zaxpyi</code> Specifies the scalar a .
x	REAL for <code>saxpyi</code> DOUBLE PRECISION for <code>daxpyi</code> COMPLEX for <code>caxpyi</code> DOUBLE COMPLEX for <code>zaxpyi</code> Array, size at least nz .
$indx$	INTEGER. Specifies the indices for the elements of x . Array, size at least nz .
y	REAL for <code>saxpyi</code> DOUBLE PRECISION for <code>daxpyi</code> COMPLEX for <code>caxpyi</code> DOUBLE COMPLEX for <code>zaxpyi</code> Array, size at least $\max(indx(i))$.

Output Parameters

y	Contains the updated vector y .
-----	-----------------------------------

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `axpyi` interface are the following:

x	Holds the vector with the number of elements nz .
$indx$	Holds the vector with the number of elements nz .
y	Holds the vector with the number of elements nz .
a	The default value is 1.

?doti

Computes the dot product of a compressed sparse real vector by a full-storage real vector.

Syntax**FORTRAN 77:**

```
res = sdoti(nz, x, indx, y)
res = ddoti(nz, x, indx, y)
```

Fortran 95:

```
res = doti(x, indx, y)
```

C:

```
float cblas_sdoti (const MKL_INT nz, const float *x, const MKL_INT *indx, const float *y);
double cblas_ddot (const MKL_INT nz, const double *x, const MKL_INT *indx, const double *y);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?doti routines return the dot product of x and y defined as

```
res = x(1)*y(indx(1)) + x(2)*y(indx(2)) + ... + x(nz)*y(indx(nz))
```

where the triple $(nz, x, indx)$ defines a sparse real vector stored in compressed form, and y is a real vector in full storage form. The functions reference only the elements of y whose indices are listed in the array $indx$. The values in $indx$ must be distinct.

Input Parameters

<i>nz</i>	INTEGER. The number of elements in x and $indx$.
<i>x</i>	REAL for sdoti DOUBLE PRECISION for ddoti Array, size at least nz .
<i>indx</i>	INTEGER. Specifies the indices for the elements of x . Array, size at least nz .
<i>y</i>	REAL for sdoti DOUBLE PRECISION for ddoti Array, size at least $\max(indx(i))$.

Output Parameters

<i>res</i>	REAL for <code>sdoti</code> DOUBLE PRECISION for <code>ddot</code> Contains the dot product of <i>x</i> and <i>y</i> , if <i>nz</i> is positive. Otherwise, <i>res</i> contains 0.
------------	--

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `doti` interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>nz</i> .
<i>indx</i>	Holds the vector with the number of elements <i>nz</i> .
<i>y</i>	Holds the vector with the number of elements <i>nz</i> .

?dotci

Computes the conjugated dot product of a compressed sparse complex vector with a full-storage complex vector.

Syntax

FORTRAN 77:

```
res = cdotci(nz, x, indx, y)
res = zdotci(nzz, x, indx, y)
```

Fortran 95:

```
res = dotci(x, indx, y)
```

C:

```
void cblas_cdotci_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void *y, void *dotui);
void cblas_zdotci_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void *y, void *dotui);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `blas.f90`
- C: `mkl.h`

Description

The `?dotci` routines return the dot product of *x* and *y* defined as

```
conjg(x(1))*y(indx(1)) + ... + conjg(x(nz))*y(indx(nz))
```

where the triple (nz, x, indx) defines a sparse complex vector stored in compressed form, and y is a real vector in full storage form. The functions reference only the elements of y whose indices are listed in the array indx . The values in indx must be distinct.

Input Parameters

nz	INTEGER. The number of elements in x and indx .
x	COMPLEX for <code>cdotci</code> DOUBLE COMPLEX for <code>zdotci</code> Array, size at least nz .
indx	INTEGER. Specifies the indices for the elements of x . Array, size at least nz .
y	COMPLEX for <code>cdotci</code> DOUBLE COMPLEX for <code>zdotci</code> Array, size at least $\max(\text{indx}(i))$.

Output Parameters

res	COMPLEX for <code>cdotci</code> DOUBLE COMPLEX for <code>zdotci</code> Contains the conjugated dot product of x and y , if nz is positive. Otherwise, it contains 0.
--------------	--

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `dotci` interface are the following:

x	Holds the vector with the number of elements (nz).
indx	Holds the vector with the number of elements (nz).
y	Holds the vector with the number of elements (nz).

?dotui

Computes the dot product of a compressed sparse complex vector by a full-storage complex vector.

Syntax

FORTRAN 77:

```
res = cdotui(nz, x, indx, y )
res = zdotui(nzz, x, indx, y )
```

Fortran 95:

```
res = dotui(x, indx, y)
```

C:

```
void cblas_cdotui_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void
*y, void *dotui);

void cblas_zdotui_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void
*y, void *dotui);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?dotui routines return the dot product of x and y defined as

```
res = x(1)*y(indx(1)) + x(2)*y(indx(2)) +...+ x(nz)*y(indx(nz))
```

where the triple (nz , x , $indx$) defines a sparse complex vector stored in compressed form, and y is a real vector in full storage form. The functions reference only the elements of y whose indices are listed in the array $indx$. The values in $indx$ must be distinct.

Input Parameters

<i>nz</i>	INTEGER. The number of elements in x and $indx$.
<i>x</i>	COMPLEX for cdotui DOUBLE COMPLEX for zdotui Array, size at least nz .
<i>indx</i>	INTEGER. Specifies the indices for the elements of x . Array, size at least nz .
<i>y</i>	COMPLEX for cdotui DOUBLE COMPLEX for zdotui Array, size at least $\max(indx(i))$.

Output Parameters

<i>res</i>	COMPLEX for cdotui DOUBLE COMPLEX for zdotui Contains the dot product of x and y , if nz is positive. Otherwise, <i>res</i> contains 0.
------------	---

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `dotui` interface are the following:

<i>x</i>	Holds the vector with the number of elements nz .
----------	---

<i>indx</i>	Holds the vector with the number of elements <i>nz</i> .
<i>y</i>	Holds the vector with the number of elements <i>nz</i> .

?gthr

Gathers a full-storage sparse vector's elements into compressed form.

Syntax

FORTRAN 77:

```
call sgthr(nz, y, x, indx )
call dgthr(nz, y, x, indx )
call cgthr(nz, y, x, indx )
call zgthr(nz, y, x, indx )
```

Fortran 95:

```
res = gthr(x, indx, y)
```

C:

```
void cblas_sgthr (const MKL_INT nz, const float *y, float *x, const MKL_INT *indx);
void cblas_dgthr (const MKL_INT nz, const double *y, double *x, const MKL_INT *indx);
void cblas_cgthr (const MKL_INT nz, const void *y, void *x, const MKL_INT *indx);
void cblas_zgthr (const MKL_INT nz, const void *y, void *x, const MKL_INT *indx);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?gthr routines gather the specified elements of a full-storage sparse vector *y* into compressed form(*nz*, *x*, *indx*). The routines reference only the elements of *y* whose indices are listed in the array *indx*:

x(*i*) = *y*(*indx*(*i*)), for *i*=1,2,..., *nz*.

Input Parameters

<i>nz</i>	INTEGER. The number of elements of <i>y</i> to be gathered.
<i>indx</i>	INTEGER. Specifies indices of elements to be gathered. Array, size at least <i>nz</i> .
<i>y</i>	REAL for sgthr DOUBLE PRECISION for dgthr COMPLEX for cgthr DOUBLE COMPLEX for zgthr

Array, size at least $\max(\text{indx}(i))$.

Output Parameters

<code>x</code>	REAL for sgthr DOUBLE PRECISION for dgthr COMPLEX for cgthr DOUBLE COMPLEX for zgthr Array, size at least nz . Contains the vector converted to the compressed form.
----------------	---

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gthr` interface are the following:

<code>x</code>	Holds the vector with the number of elements nz .
<code>indx</code>	Holds the vector with the number of elements nz .
<code>y</code>	Holds the vector with the number of elements nz .

?gthrz

Gathers a sparse vector's elements into compressed form, replacing them by zeros.

Syntax

FORTRAN 77:

```
call sgthrz( $nz$ ,  $y$ ,  $x$ ,  $indx$ )
call dgthrz( $nz$ ,  $y$ ,  $x$ ,  $indx$ )
call cgthrz( $nz$ ,  $y$ ,  $x$ ,  $indx$ )
call zgthrz( $nz$ ,  $y$ ,  $x$ ,  $indx$ )
```

Fortran 95:

```
res = gthrz( $x$ ,  $indx$ ,  $y$ )
```

C:

```
void cblas_sgthrz (const MKL_INT  $nz$ , float * $y$ , float * $x$ , const MKL_INT * $indx$ );
void cblas_dgthrz (const MKL_INT  $nz$ , double * $y$ , double * $x$ , const MKL_INT * $indx$ );
void cblas_cgthrz (const MKL_INT  $nz$ , void * $y$ , void * $x$ , const MKL_INT * $indx$ );
void cblas_zgthrz (const MKL_INT  $nz$ , void * $y$ , void * $x$ , const MKL_INT * $indx$ );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90

- C: mkl.h

Description

The `?gthrz` routines gather the elements with indices specified by the array *indx* from a full-storage vector *y* into compressed form (*nz*, *x*, *indx*) and overwrite the gathered elements of *y* by zeros. Other elements of *y* are not referenced or modified (see also [?gthr](#)).

Input Parameters

<i>nz</i>	INTEGER. The number of elements of <i>y</i> to be gathered.
<i>indx</i>	INTEGER. Specifies indices of elements to be gathered. Array, size at least <i>nz</i> .
<i>y</i>	REAL for sgthrz DOUBLE PRECISION for dgthrz COMPLEX for cgthrz DOUBLE COMPLEX for zgthrz Array, size at least $\max(indx(i))$.

Output Parameters

<i>x</i>	REAL for sgthrz DOUBLE PRECISION for dgthrz COMPLEX for cgthrz DOUBLE COMPLEX for zgthrz Array, size at least <i>nz</i> . Contains the vector converted to the compressed form.
<i>y</i>	The updated vector <i>y</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gthrz` interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>nz</i> .
<i>indx</i>	Holds the vector with the number of elements <i>nz</i> .
<i>y</i>	Holds the vector with the number of elements <i>nz</i> .

?roti

Applies Givens rotation to sparse vectors one of which is in compressed form.

Syntax

FORTRAN 77:

```
call sroti(nz, x, indx, y, c, s)
call droti(nz, x, indx, y, c, s)
```

Fortran 95:

```
call roti(x, indx, y, c, s)
```

C:

```
void cblas_sroti (const MKL_INT nz, float *x, const MKL_INT *indx, float *y, const
float c, const float s);
void cblas_droti (const MKL_INT nz, double *x, const MKL_INT *indx, double *y, const
double c, const double s);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?roti routines apply the Givens rotation to elements of two real vectors, x (in compressed form nz , x , $indx$) and y (in full storage form):

```
x(i) = c*x(i) + s*y(indx(i))
```

```
y(indx(i)) = c*y(indx(i))- s*x(i)
```

The routines reference only the elements of y whose indices are listed in the array $indx$. The values in $indx$ must be distinct.

Input Parameters

<i>nz</i>	INTEGER. The number of elements in x and $indx$.
<i>x</i>	REAL for sroti DOUBLE PRECISION for droti Array, size at least nz .
<i>indx</i>	INTEGER. Specifies the indices for the elements of x . Array, size at least nz .
<i>y</i>	REAL for sroti DOUBLE PRECISION for droti Array, size at least $\max(indx(i))$.
<i>c</i>	REAL for sroti DOUBLE PRECISION for droti. A scalar.
<i>s</i>	REAL for sroti

DOUBLE PRECISION **for** droti.

A scalar.

Output Parameters

<i>x</i> and <i>y</i>	The updated arrays.
-----------------------	---------------------

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *roti* interface are the following:

<i>x</i>	Holds the vector with the number of elements <i>nz</i> .
<i>indx</i>	Holds the vector with the number of elements <i>nz</i> .
<i>y</i>	Holds the vector with the number of elements <i>nz</i> .

?sctr

Converts compressed sparse vectors into full storage form.

Syntax

FORTRAN 77:

```
call ssctr(nz, x, indx, y )
call dsctr(nz, x, indx, y )
call csctr(nz, x, indx, y )
call zsctr(nz, x, indx, y )
```

Fortran 95:

```
call sctr(x, indx, y)
```

C:

```
void cblas_ssctr (const MKL_INT nz, const float *x, const MKL_INT *indx, float *y);
void cblas.dsctr (const MKL_INT nz, const double *x, const MKL_INT *indx, double *y);
void cblas.csctr (const MKL_INT nz, const void *x, const MKL_INT *indx, void *y);
void cblas_zsctr (const MKL_INT nz, const void *x, const MKL_INT *indx, void *y);
```

Include Files

- **Fortran:** mkl.fi
- **Fortran 95:** blas.f90
- **C:** mkl.h

Description

The ?sctr routines scatter the elements of the compressed sparse vector (*nz*, *x*, *indx*) to a full-storage vector *y*. The routines modify only the elements of *y* whose indices are listed in the array *indx*:

$y(indx(i)) = x(i)$, for $i=1, 2, \dots, nz$.

Input Parameters

nz	INTEGER. The number of elements of x to be scattered.
$indx$	INTEGER. Specifies indices of elements to be scattered. Array, size at least nz .
x	REAL for ssctr DOUBLE PRECISION for dsctr COMPLEX for csctr DOUBLE COMPLEX for zsctr Array, size at least nz . Contains the vector to be converted to full-storage form.

Output Parameters

y	REAL for ssctr DOUBLE PRECISION for dsctr COMPLEX for csctr DOUBLE COMPLEX for zsctr Array, size at least $\max(indx(i))$. Contains the vector y with updated elements.
-----	---

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sctr` interface are the following:

x	Holds the vector with the number of elements nz .
$indx$	Holds the vector with the number of elements nz .
y	Holds the vector with the number of elements nz .

Sparse BLAS Level 2 and Level 3 Routines

This section describes Sparse BLAS Level 2 and Level 3 routines included in the Intel® Math Kernel Library (Intel® MKL) . Sparse BLAS Level 2 is a group of routines and functions that perform operations between a sparse matrix and dense vectors. Sparse BLAS Level 3 is a group of routines and functions that perform operations between a sparse matrix and dense matrices.

The terms and concepts required to understand the use of the Intel MKL Sparse BLAS Level 2 and Level 3 routines are discussed in the [Linear Solvers Basics](#) appendix.

The Sparse BLAS routines can be useful to implement iterative methods for solving large sparse systems of equations or eigenvalue problems. For example, these routines can be considered as building blocks for [Iterative Sparse Solvers based on Reverse Communication Interface \(RCI ISS\)](#) described in the Chapter 8 of the manual.

Intel MKL provides Sparse BLAS Level 2 and Level 3 routines with typical (or conventional) interface similar to the interface used in the NIST* Sparse BLAS library [Rem05].

Some software packages and libraries (the PARDISO* Solver used in Intel MKL, Sparskit 2 [Saad94], the Compaq* Extended Math Library (CXML)[CXML01]) use different (early) variation of the compressed sparse row (CSR) format and support only Level 2 operations with simplified interfaces. Intel MKL provides an additional set of Sparse BLAS Level 2 routines with similar simplified interfaces. Each of these routines operates only on a matrix of the fixed type.

The routines described in this section support both one-based indexing and zero-based indexing of the input data (see details in the section [One-based and Zero-based Indexing](#)).

Naming Conventions in Sparse BLAS Level 2 and Level 3

Each Sparse BLAS Level 2 and Level 3 routine has a six- or eight-character base name preceded by the prefix `mkl_` or `mkl_cspblas_`.

The routines with typical (conventional) interface have six-character base names in accordance with the template:

```
mkl_<character> <data> <operation>()
```

The routines with simplified interfaces have eight-character base names in accordance with the templates:

```
mkl_<character> <data> <mtype> <operation>()
```

for routines with one-based indexing; and

```
mkl_cspblas_<character> <data> <mtype> <operation>()
```

for routines with zero-based indexing.

The `<character>` field indicates the data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision

The `<data>` field indicates the sparse matrix storage format (see section [Sparse Matrix Storage Formats](#)):

coo	coordinate format
csr	compressed sparse row format and its variations
csc	compressed sparse column format and its variations
dia	diagonal format
sky	skyline storage format
bsr	block sparse row format and its variations

The `<operation>` field indicates the type of operation:

mv	matrix-vector product (Level 2)
mm	matrix-matrix product (Level 3)
sv	solving a single triangular system (Level 2)
sm	solving triangular systems with multiple right-hand sides (Level 3)

The field `<mtype>` indicates the matrix type:

ge	sparse representation of a general matrix
sy	sparse representation of the upper or lower triangle of a symmetric matrix
tr	sparse representation of a triangular matrix

Sparse Matrix Storage Formats

The current version of Intel MKL Sparse BLAS Level 2 and Level 3 routines support the following point entry [Duff86] storage formats for sparse matrices:

- *compressed sparse row* format (CSR) and its variations;
- *compressed sparse column* format (CSC);
- *coordinate* format;
- *diagonal* format;
- *skyline* storage format;

and one block entry storage format:

- *block sparse row* format (BSR) and its variations.

For more information see "[Sparse Matrix Storage Formats](#)" in Appendix A.

Intel MKL provides auxiliary routines - [matrix converters](#) - that convert sparse matrix from one storage format to another.

Routines and Supported Operations

This section describes operations supported by the Intel MKL Sparse BLAS Level 2 and Level 3 routines. The following notations are used here:

A is a sparse matrix;
B and *C* are dense matrices;
D is a diagonal scaling matrix;
x and *y* are dense vectors;
alpha and *beta* are scalars;

op (A) is one of the possible operations:

```
op (A) = A;
op (A) = A' - transpose of A;
op (A) = conj (A') - conjugated transpose of A.
```

inv (op (A)) denotes the inverse of *op (A)*.

The Intel MKL Sparse BLAS Level 2 and Level 3 routines support the following operations:

- computing the vector product between a sparse matrix and a dense vector:

```
y := alpha*op(A)*x + beta*y
```

- solving a single triangular system:

```
y := alpha*inv(op(A))*x
```

- computing a product between sparse matrix and dense matrix:

```
C := alpha*op(A)*B + beta*C
```

- solving a sparse triangular system with multiple right-hand sides:

```
C := alpha*inv(op(A))*B
```

Intel MKL provides an additional set of the Sparse BLAS Level 2 routines with *simplified interfaces*. Each of these routines operates on a matrix of the fixed type. The following operations are supported:

- computing the vector product between a sparse matrix and a dense vector (for general and symmetric matrices):

```
y := op(A) * x
```

- solving a single triangular system (for triangular matrices):

```
y := inv(op(A)) * x
```

Matrix type is indicated by the field *<mtype>* in the routine name (see section [Naming Conventions in Sparse BLAS Level 2 and Level 3](#)).

NOTE

The routines with simplified interfaces support only four sparse matrix storage formats, specifically:

CSR format in the 3-array variation accepted in the direct sparse solvers and in the CXML;
diagonal format accepted in the CXML;
coordinate format;
BSR format in the 3-array variation.

Note that routines with both typical (conventional) and simplified interfaces use the same computational kernels that work with certain internal data structures.

The Intel MKL Sparse BLAS Level 2 and Level 3 routines do not support in-place operations.

Complete list of all routines is given in the ["Sparse BLAS Level 2 and Level 3 Routines"](#).

Interface Consideration

One-Based and Zero-Based Indexing

The Intel MKL Sparse BLAS Level 2 and Level 3 routines support one-based and zero-based indexing of data arrays.

Routines with typical interfaces support zero-based indexing for the following sparse data storage formats: CSR, CSC, BSR, and COO. Routines with simplified interfaces support zero based indexing for the following sparse data storage formats: CSR, BSR, and COO. See the complete list of [Sparse BLAS Level 2 and Level 3 Routines](#).

The one-based indexing uses the convention of starting array indices at 1. The zero-based indexing uses the convention of starting array indices at 0. For example, indices of the 5-element array *x* can be presented in case of one-based indexing as follows:

Element index: 1 2 3 4 5

Element value: 1.0 5.0 7.0 8.0 9.0

and in case of zero-based indexing as follows:

Element index: 0 1 2 3 4

Element value: 1.0 5.0 7.0 8.0 9.0

The detailed descriptions of the one-based and zero-based variants of the sparse data storage formats are given in the ["Sparse Matrix Storage Formats"](#) in Appendix A.

Most parameters of the routines are identical for both one-based and zero-based indexing, but some of them have certain differences. The following table lists all these differences.

Parameter	One-based Indexing	Zero-based Indexing
<i>val</i>	Array containing non-zero elements of the matrix A , its length is $pntre(m) - pntrb(1)$.	Array containing non-zero elements of the matrix A , its length is $pntre(m-1) - pntrb(0)$.
<i>pntrb</i>	Array of length m . This array contains row indices, such that $pntrb(i) - pntrb(1)+1$ is the first index of row i in the arrays <i>val</i> and <i>indx</i>	Array of length m . This array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of row i in the arrays <i>val</i> and <i>indx</i> .
<i>pntre</i>	Array of length m . This array contains row indices, such that $pntre(I) - pntrb(1)$ is the last index of row i in the arrays <i>val</i> and <i>indx</i> .	Array of length m . This array contains row indices, such that $pntre(i) - pntrb(0)-1$ is the last index of row i in the arrays <i>val</i> and <i>indx</i> .
<i>ia</i>	Array of length $m + 1$, containing indices of elements in the array <i>a</i> , such that $ia(i)$ is the index in the array <i>a</i> of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zeros plus one.	Array of length $m+1$, containing indices of elements in the array <i>a</i> , such that $ia(i)$ is the index in the array <i>a</i> of the first non-zero element from the row i . The value of the last element $ia(m)$ is equal to the number of non-zeros.
<i>ldb</i>	Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program.	Specifies the second dimension of <i>b</i> as declared in the calling (sub)program.
<i>ldc</i>	Specifies the leading dimension of <i>c</i> as declared in the calling (sub)program.	Specifies the second dimension of <i>c</i> as declared in the calling (sub)program.

Difference Between Fortran and C Interfaces

Intel MKL provides both Fortran and C interfaces to all Sparse BLAS Level 2 and Level 3 routines. Parameter descriptions are common for both interfaces with the exception of data types that refer to the FORTRAN 77 standard types. Correspondence between data types specific to the Fortran and C interfaces are given below:

Fortran	C
REAL*4	float
REAL*8	double
INTEGER*4	int
INTEGER*8	long long int
CHARACTER	char

For routines with C interfaces all parameters (including scalars) must be passed by references.

Another difference is how two-dimensional arrays are represented. In Fortran the column-major order is used, and in C row-major order. This changes the meaning of the parameters *ldb* and *ldc* (see the table above).

Differences Between Intel MKL and NIST* Interfaces

The Intel MKL Sparse BLAS Level 3 routines have the following conventional interfaces:

`mkl_xyyyymm(transa, m, n, k, alpha, matdescra, arg(A), b, ldb, beta, c, ldc)`, for matrix-matrix product;

`mkl_xyyysm(transa, m, n, alpha, matdescra, arg(A), b, ldb, c, ldc)`, for triangular solvers with multiple right-hand sides.

Here `x` denotes data type, and `yyy` - sparse matrix data structure (storage format).

The analogous NIST* Sparse BLAS (NSB) library routines have the following interfaces:

`xyyymm(transa, m, n, k, alpha, descra, arg(A), b, ldb, beta, c, ldc, work, lwork)`, for matrix-matrix product;

`xyyyssm(transa, m, n, unitd, dv, alpha, descra, arg(A), b, ldb, beta, c, ldc, work, lwork)`, for triangular solvers with multiple right-hand sides.

Some similar arguments are used in both libraries. The argument `transa` indicates what operation is performed and is slightly different in the NSB library (see [Table "Parameter `transa`"](#)). The arguments `m` and `k` are the number of rows and column in the matrix `A`, respectively, `n` is the number of columns in the matrix `C`. The arguments `alpha` and `beta` are scalar `alpha` and `beta` respectively (`beta` is not used in the Intel MKL triangular solvers.) The arguments `b` and `c` are rectangular arrays with the leading dimension `ldb` and `ldc`, respectively. `arg(A)` denotes the list of arguments that describe the sparse representation of `A`.

Parameter `transa`

	MKL interface	NSB interface	Operation
data type	CHARACTER*1	INTEGER	
value	N or n	0	$\text{op}(A) = A$
	T or t	1	$\text{op}(A) = A'$
	C or c	2	$\text{op}(A) = A'$

Parameter `matdescra`

The parameter `matdescra` describes the relevant characteristic of the matrix `A`. This manual describes `matdescra` as an array of six elements in line with the NIST* implementation. However, only the first four elements of the array are used in the current versions of the Intel MKL Sparse BLAS routines. Elements `matdescra(5)` and `matdescra(6)` are reserved for future use. Note that whether `matdescra` is described in your application as an array of length 6 or 4 is of no importance because the array is declared as a pointer in the Intel MKL routines. To learn more about declaration of the `matdescra` array, see the Sparse BLAS examples located in the Intel MKL installation directory: `examples/spblasf/` for Fortran or `examples/spblasf/` for C. The table below lists elements of the parameter `matdescra`, their for Fortran values and meanings. The parameter `matdescra` corresponds to the argument `descra` from NSB library.

Possible Values of the Parameter `matdescra` (`descra`)

	MKL interface	NSB interface	Matrix characteristics	
	one-based indexing	zero-based indexing		
data type	CHARACTER	Char	INTEGER	
1st element	<code>matdescra(1)</code>	<code>matdescra(0)</code>	<code>descra(1)</code>	matrix structure
value	G	G	0	general
	S	S	1	symmetric ($A = A'$)
	H	H	2	Hermitian ($A = \text{conjg}(A')$)

	MKL interface		NSB interface	Matrix characteristics
	T	T	3	triangular
	A	A	4	skew(anti)-symmetric ($A=-A'$)
	D	D	5	diagonal
2nd element value	<i>matdescra</i> (2)	<i>matdescra</i> (1)	<i>descra</i> (2)	upper/lower triangular indicator
	L	L	1	lower
	U	U	2	upper
3rd element value	<i>matdescra</i> (3)	<i>matdescra</i> (2)	<i>descra</i> (3)	main diagonal type
	N	N	0	non-unit
	U	U	1	unit
4th element value	<i>matdescra</i> (4)	<i>matdescra</i> (3)		type of indexing
	F			one-based indexing
	C			zero-based indexing

In some cases possible element values of the parameter *matdescra* depend on the values of other elements. The [Table "Possible Combinations of Element Values of the Parameter *matdescra*"](#) lists all possible combinations of element values for both multiplication routines and triangular solvers.

Possible Combinations of Element Values of the Parameter *matdescra*

Routines	matdescra(1)	matdescra(2)	matdescra(3)	matdescra(4)
Multiplication Routines	G	ignored	ignored	F (default) or C
	S or H	L (default)	N (default)	F (default) or C
	S or H	L (default)	U	F (default) or C
	S or H	U	N (default)	F (default) or C
	S or H	U	U	F (default) or C
	A	L (default)	ignored	F (default) or C
	A	U	ignored	F (default) or C
Multiplication Routines and Triangular Solvers	T	L	U	F (default) or C
	T	L	N	F (default) or C
	T	U	U	F (default) or C
	T	U	N	F (default) or C
	D	ignored	N (default)	F (default) or C
	D	ignored	U	F (default) or C

For a matrix in the skyline format with the main diagonal declared to be a unit, diagonal elements must be stored in the sparse representation even if they are zero. In all other formats, diagonal elements can be stored (if needed) in the sparse representation if they are not zero.

Operations with Partial Matrices

One of the distinctive feature of the Intel MKL Sparse BLAS routines is a possibility to perform operations only on partial matrices composed of certain parts (triangles and the main diagonal) of the input sparse matrix. It can be done by setting properly first three elements of the parameter *matdescra*.

An arbitrary sparse matrix A can be decomposed as

$$A = L + D + U$$

where L is the strict lower triangle of A , U is the strict upper triangle of A , D is the main diagonal.

[Table "Output Matrices for Multiplication Routines"](#) shows correspondence between the output matrices and values of the parameter *matdescra* for the sparse matrix A for multiplication routines.

Output Matrices for Multiplication Routines

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
G	ignored	ignored	$\alpha * \text{op}(A) * x + \beta * y$ $\alpha * \text{op}(A) * B + \beta * C$
S or H	L	N	$\alpha * \text{op}(L+D+L') * x + \beta * y$ $\alpha * \text{op}(L+D+L') * B + \beta * C$
S or H	L	U	$\alpha * \text{op}(L+I+L') * x + \beta * y$ $\alpha * \text{op}(L+I+L') * B + \beta * C$
S or H	U	N	$\alpha * \text{op}(U'+D+U) * x + \beta * y$ $\alpha * \text{op}(U'+D+U) * B + \beta * C$
S or H	U	U	$\alpha * \text{op}(U'+I+U) * x + \beta * y$ $\alpha * \text{op}(U'+I+U) * B + \beta * C$
T	L	U	$\alpha * \text{op}(L+I) * x + \beta * y$ $\alpha * \text{op}(L+I) * B + \beta * C$
T	L	N	$\alpha * \text{op}(L+D) * x + \beta * y$ $\alpha * \text{op}(L+D) * B + \beta * C$
T	U	U	$\alpha * \text{op}(U+I) * x + \beta * y$ $\alpha * \text{op}(U+I) * B + \beta * C$
T	U	N	$\alpha * \text{op}(U+D) * x + \beta * y$ $\alpha * \text{op}(U+D) * B + \beta * C$
A	L	ignored	$\alpha * \text{op}(L-L') * x + \beta * y$ $\alpha * \text{op}(L-L') * B + \beta * C$
A	U	ignored	$\alpha * \text{op}(U-U') * x + \beta * y$ $\alpha * \text{op}(U-U') * B + \beta * C$
D	ignored	N	$\alpha * D * x + \beta * y$ $\alpha * D * B + \beta * C$
D	ignored	U	$\alpha * x + \beta * y$ $\alpha * B + \beta * C$

[Table "Output Matrices for Triangular Solvers"](#) shows correspondence between the output matrices and values of the parameter *matdescra* for the sparse matrix A for triangular solvers.

Output Matrices for Triangular Solvers

matdescra(1)	matdescra(2)	matdescra(3)	Output Matrix
T	L	N	$\alpha^{-1} \operatorname{op}(L+L)^{-1} x$ $\alpha^{-1} \operatorname{op}(L+L)^{-1} B$
T	L	U	$\alpha^{-1} \operatorname{op}(L+L)^{-1} x$ $\alpha^{-1} \operatorname{op}(L+L)^{-1} B$
T	U	N	$\alpha^{-1} \operatorname{op}(U+U)^{-1} x$ $\alpha^{-1} \operatorname{op}(U+U)^{-1} B$
T	U	U	$\alpha^{-1} \operatorname{op}(U+U)^{-1} x$ $\alpha^{-1} \operatorname{op}(U+U)^{-1} B$
D	ignored	N	$\alpha^{-1} D^{-1} x$ $\alpha^{-1} D^{-1} B$
D	ignored	U	$\alpha^{-1} x$ $\alpha^{-1} B$

Sparse BLAS Level 2 and Level 3 Routines.

Table “Sparse BLAS Level 2 and Level 3 Routines” lists the sparse BLAS Level 2 and Level 3 routines described in more detail later in this section.

Sparse BLAS Level 2 and Level 3 Routines

Routine/Function	Description
Simplified interface, one-based indexing	
<code>mkl_?csrgemv</code>	Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation)
<code>mkl_?bsrgemv</code>	Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation).
<code>mkl_?coogemv</code>	Computes matrix - vector product of a sparse general matrix in the coordinate format.
<code>mkl_?diagemv</code>	Computes matrix - vector product of a sparse general matrix in the diagonal format.
<code>mkl_?csrsymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation)
<code>mkl_?bsrsymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation).
<code>mkl_?coosymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format.
<code>mkl_?diasymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the diagonal format.
<code>mkl_?csrtrsv</code>	Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation).

Routine/Function	Description
<code>mkl_?bsrtrs</code>	Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation).
<code>mkl_?cootrs</code>	Triangular solvers with simplified interface for a sparse matrix in the coordinate format.
<code>mkl_?diatrs</code>	Triangular solvers with simplified interface for a sparse matrix in the diagonal format.

Simplified interface, zero-based indexing

<code>mkl_cspblas_?csrgemv</code>	Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?bsrgemv</code>	Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?coogemv</code>	Computes matrix - vector product of a sparse general matrix in the coordinate format with zero-based indexing.
<code>mkl_cspblas_?csrsymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation) with zero-based indexing
<code>mkl_cspblas_?bsrsymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?coosymv</code>	Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format with zero-based indexing.
<code>mkl_cspblas_?csrtrs</code>	Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?bsrtrs</code>	Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation) with zero-based indexing.
<code>mkl_cspblas_?cootrs</code>	Triangular solver with simplified interface for a sparse matrix in the coordinate format with zero-based indexing.

Typical (conventional) interface, one-based and zero-based indexing

<code>mkl_?csrmv</code>	Computes matrix - vector product of a sparse matrix in the CSR format.
<code>mkl_?bsrmv</code>	Computes matrix - vector product of a sparse matrix in the BSR format.
<code>mkl_?cscmv</code>	Computes matrix - vector product for a sparse matrix in the CSC format.
<code>mkl_?coomv</code>	Computes matrix - vector product for a sparse matrix in the coordinate format.
<code>mkl_?csrs</code>	Solves a system of linear equations for a sparse matrix in the CSR format.

Routine/Function	Description
<code>mkl_?bsrsv</code>	Solves a system of linear equations for a sparse matrix in the BSR format.
<code>mkl_?cscsv</code>	Solves a system of linear equations for a sparse matrix in the CSC format.
<code>mkl_?coosv</code>	Solves a system of linear equations for a sparse matrix in the coordinate format.
<code>mkl_?csrmm</code>	Computes matrix - matrix product of a sparse matrix in the CSR format
<code>mkl_?bsrmm</code>	Computes matrix - matrix product of a sparse matrix in the BSR format.
<code>mkl_?cscmm</code>	Computes matrix - matrix product of a sparse matrix in the CSC format
<code>mkl_?coomm</code>	Computes matrix - matrix product of a sparse matrix in the coordinate format.
<code>mkl_?csrsm</code>	Solves a system of linear matrix equations for a sparse matrix in the CSR format.
<code>mkl_?bsrsm</code>	Solves a system of linear matrix equations for a sparse matrix in the BSR format.
<code>mkl_?cscsm</code>	Solves a system of linear matrix equations for a sparse matrix in the CSC format.
<code>mkl_?coosm</code>	Solves a system of linear matrix equations for a sparse matrix in the coordinate format.

Typical (conventional) interface, one-based indexing

<code>mkl_?diamv</code>	Computes matrix - vector product of a sparse matrix in the diagonal format.
<code>mkl_?skymv</code>	Computes matrix - vector product for a sparse matrix in the skyline storage format.
<code>mkl_?diasv</code>	Solves a system of linear equations for a sparse matrix in the diagonal format.
<code>mkl_?skysv</code>	Solves a system of linear equations for a sparse matrix in the skyline format.
<code>mkl_?diamm</code>	Computes matrix - matrix product of a sparse matrix in the diagonal format.
<code>mkl_?skymm</code>	Computes matrix - matrix product of a sparse matrix in the skyline storage format.
<code>mkl_?diasm</code>	Solves a system of linear matrix equations for a sparse matrix in the diagonal format.
<code>mkl_?skysm</code>	Solves a system of linear matrix equations for a sparse matrix in the skyline storage format.

Auxiliary routines

Matrix converters

Routine/Function	Description
<code>mkl_?dnsCSR</code>	Converts a sparse matrix in uncompressed representation to CSR format (3-array variation) and vice versa.
<code>mkl_?csRCOO</code>	Converts a sparse matrix in CSR format (3-array variation) to coordinate format and vice versa.
<code>mkl_?csRBsr</code>	Converts a sparse matrix in CSR format to BSR format (3-array variations) and vice versa.
<code>mkl_?csRCsc</code>	Converts a sparse matrix in CSR format to CSC format and vice versa (3-array variations).
<code>mkl_?csRDia</code>	Converts a sparse matrix in CSR format (3-array variation) to diagonal format and vice versa.
<code>mkl_?csRSky</code>	Converts a sparse matrix in CSR format (3-array variation) to sky line format and vice versa.
Operations on sparse matrices	
<code>mkl_?csRadd</code>	Computes the sum of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.
<code>mkl_?csRMultCSR</code>	Computes the product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.
<code>mkl_?csRMultD</code>	Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing. The result is stored in the dense matrix.

`mkl_?csrgemv`

Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_scsrgeMV(transa, m, a, ia, ja, x, y)
call mkl_dcsrgeMV(transa, m, a, ia, ja, x, y)
call mkl_ccsrgeMV(transa, m, a, ia, ja, x, y)
call mkl_zcsrgeMV(transa, m, a, ia, ja, x, y)
```

C:

```
void mkl_scsrgeMV (char *transa, MKL_INT *m, float *a, MKL_INT *ia, MKL_INT *ja, float *x, float *y);
void mkl_dcsrgeMV (char *transa, MKL_INT *m, double *a, MKL_INT *ia, MKL_INT *ja, double *x, double *y);
void mkl_ccsrgeMV (char *transa, MKL_INT *m, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zcsrgeMV (char *transa, MKL_INT *m, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?csrgemv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A'*x,
```

where:

x and *y* are vectors,

A is an *m*-by-*m* sparse square matrix in the CSR format (3-array variation), *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then as <i>y</i> := <i>A</i> * <i>x</i> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>y</i> := <i>A'</i> * <i>x</i> ,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>a</i>	REAL for <code>mkl_scsrgemv</code> . DOUBLE PRECISION for <code>mkl_dcsrgemv</code> . COMPLEX for <code>mkl_ccsrgemv</code> . DOUBLE COMPLEX for <code>mkl_zcsrgemv</code> . Array containing non-zero elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	INTEGER. Array of length <i>m</i> + 1, containing indices of elements in the array <i>a</i> , such that <i>ia</i> (<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> (<i>m</i> + 1) is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i> . Its length is equal to the length of the array <i>a</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.

x

REAL for mkl_scsrgemv.
 DOUBLE PRECISION for mkl_dcsrgemv.
 COMPLEX for mkl_ccsrgemv.
 DOUBLE COMPLEX for mkl_zcsrgemv.
Array, size is m .
 On entry, the array x must contain the vector x .

Output Parameters

y

REAL for mkl_scsrgemv.
 DOUBLE PRECISION for mkl_dcsrgemv.
 COMPLEX for mkl_ccsrgemv.
 DOUBLE COMPLEX for mkl_zcsrgemv.
Array, size at least m .
 On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrgemv(transa, m, a, ia, ja, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m
```

```
  INTEGER ia(*), ja(*)
```

```
  REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcsrgemv(transa, m, a, ia, ja, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m
```

```
  INTEGER ia(*), ja(*)
```

```
  DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccsrgemv(transa, m, a, ia, ja, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m
```

```
  INTEGER ia(*), ja(*)
```

```
  COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcsrgemv(transa, m, a, ia, ja, x, y)
  CHARACTER*1 transa
  INTEGER m
  INTEGER ia(*), ja(*)
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_scsrgemv(char *transa, int *m, float *a,
int *ia, int *ja, float *x, float *y);

void mkl_dcsrgemv(char *transa, int *m, double *a,
int *ia, int *ja, double *x, double *y);

void mkl_ccsrgemv(char *transa, int *m, MKL_Complex8 *a,
int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zcsrgemv(char *transa, int *m, MKL_Complex16 *a,
int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?bsrgemv

Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with one-based indexing.

Syntax**Fortran:**

```
call mkl_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

C:

```
void mkl_sbsrgemv (char *transa, MKL_INT *m, MKL_INT *lb, float *a, MKL_INT *ia,
MKL_INT *ja, float *x, float *y);

void mkl_dbsrgemv (char *transa, MKL_INT *m, MKL_INT *lb, double *a, MKL_INT *ia,
MKL_INT *ja, double *x, double *y);

void mkl_cbsrgemv (char *transa, MKL_INT *m, MKL_INT *lb, MKL_Complex8 *a, MKL_INT *ia,
MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zbsrgemv (char *transa, MKL_INT *m, MKL_INT *lb, MKL_Complex16 *a, MKL_INT
*ia, MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- **Fortran:** mkl.fi
- **C:** mkl.h

Description

The `mkl_?bsrgemv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A'*x,
```

where:

x and y are vectors,

A is an m -by- m block sparse square matrix in the BSR format (3-array variation), A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>transa</code>	CHARACTER*1. Specifies the operation. If <code>transa</code> = ' <code>N</code> ' or ' <code>n</code> ', then the matrix-vector product is computed as <code>y := A*x</code> If <code>transa</code> = ' <code>T</code> ' or ' <code>t</code> ' or ' <code>C</code> ' or ' <code>c</code> ', then the matrix-vector product is computed as <code>y := A'*x</code> ,
<code>m</code>	INTEGER. Number of block rows of the matrix A .
<code>lb</code>	INTEGER. Size of the block in the matrix A .
<code>a</code>	REAL for <code>mkl_sbsrgemv</code> . DOUBLE PRECISION for <code>mkl_dbsrgemv</code> . COMPLEX for <code>mkl_cbsrgemv</code> . DOUBLE COMPLEX for <code>mkl_zbsrgemv</code> . Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by <code>lb*lb</code> . Refer to <code>values</code> array description in BSR Format for more details.
<code>ia</code>	INTEGER. Array of length $(m + 1)$, containing indices of block in the array <code>a</code> , such that <code>ia(i)</code> is the index in the array <code>a</code> of the first non-zero element from the row i . The value of the last element <code>ia(m + 1)</code> is equal to the number of non-zero blocks plus one. Refer to <code>rowIndex</code> array description in BSR Format for more details.
<code>ja</code>	INTEGER. Array containing the column indices for each non-zero block in the matrix A . Its length is equal to the number of non-zero blocks of the matrix A . Refer to <code>columns</code> array description in BSR Format for more details.

x REAL for mkl_sbsrgemv.
 DOUBLE PRECISION for mkl_dbsrgemv.
 COMPLEX for mkl_cbsrgemv.
 DOUBLE COMPLEX for mkl_zbsrgemv.
 Array, size ($m \times lb$).
 On entry, the array *x* must contain the vector *x*.

Output Parameters

y REAL for mkl_sbsrgemv.
 DOUBLE PRECISION for mkl_dbsrgemv.
 COMPLEX for mkl_cbsrgemv.
 DOUBLE COMPLEX for mkl_zbsrgemv.
 Array, size at least ($m \times lb$).
 On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, lb
```

```
  INTEGER ia(*), ja(*)
```

```
  REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, lb
```

```
  INTEGER ia(*), ja(*)
```

```
  DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, lb
```

```
  INTEGER ia(*), ja(*)
```

```
  COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
  CHARACTER*1 transa
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_dbsrgemv(char *transa, int *m, int *lb, double *a,
int *ia, int *ja, double *x, double *y);
void mkl_sbsrgemv(char *transa, int *m, int *lb, float *a,
int *ia, int *ja, float *x, float *y);
void mkl_cbsrgemv(char *transa, int *m, int *lb, MKL_Complex8 *a,
int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zbsrgemv(char *transa, int *m, int *lb, MKL_Complex16 *a,
int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?coogemv

Computes matrix-vector product of a sparse general matrix stored in the coordinate format with one-based indexing.

Syntax**Fortran:**

```
call mkl_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

C:

```
void mkl_scoogemv (char *transa, MKL_INT *m, float *val, MKL_INT *rowind, MKL_INT
*colind, MKL_INT *nnz, float *x, float *y);
void mkl_dcoogemv (char *transa, MKL_INT *m, double *val, MKL_INT *rowind, MKL_INT
*colind, MKL_INT *nnz, double *x, double *y);
void mkl_ccoogemv (char *transa, MKL_INT *m, MKL_Complex8 *val, MKL_INT *rowind,
MKL_INT *colind, MKL_INT *nnz, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zcoogemv (char *transa, MKL_INT *m, MKL_Complex16 *val, MKL_INT *rowind,
MKL_INT *colind, MKL_INT *nnz, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- **Fortran:** mkl.fi
- **C:** mkl.h

Description

The `mkl_?coogemv` routine performs a matrix-vector operation defined as

```
y := A*x
```

or

```
y := A'*x,
```

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the coordinate format, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>transa</code>	CHARACTER*1. Specifies the operation. If <code>transa</code> = ' <code>N</code> ' or ' <code>n</code> ', then the matrix-vector product is computed as <code>y := A*x</code> If <code>transa</code> = ' <code>T</code> ' or ' <code>t</code> ' or ' <code>C</code> ' or ' <code>c</code> ', then the matrix-vector product is computed as <code>y := A'*x</code> ,
<code>m</code>	INTEGER. Number of rows of the matrix A .
<code>val</code>	REAL for <code>mkl_scoogemv</code> . DOUBLE PRECISION for <code>mkl_dcoogemv</code> . COMPLEX for <code>mkl_ccoogemv</code> . DOUBLE COMPLEX for <code>mkl_zcoogemv</code> . Array of length nnz , contains non-zero elements of the matrix A in the arbitrary order. Refer to <code>values</code> array description in Coordinate Format for more details.
<code>rowind</code>	INTEGER. Array of length nnz , contains the row indices for each non-zero element of the matrix A . Refer to <code>rows</code> array description in Coordinate Format for more details.
<code>colind</code>	INTEGER. Array of length nnz , contains the column indices for each non-zero element of the matrix A . Refer to <code>columns</code> array description in Coordinate Format for more details.
<code>nnz</code>	INTEGER. Specifies the number of non-zero element of the matrix A . Refer to <code>nnz</code> description in Coordinate Format for more details.
<code>x</code>	REAL for <code>mkl_scoogemv</code> .

DOUBLE PRECISION **for** mkl_dcoogemv.
 COMPLEX **for** mkl_ccoogemv.
 DOUBLE COMPLEX **for** mkl_zcoogemv.
Array, size is m .
 One entry, the array x must contain the vector x .

Output Parameters

y REAL **for** mkl_scoogemv.
 DOUBLE PRECISION **for** mkl_dcoogemv.
 COMPLEX **for** mkl_ccoogemv.
 DOUBLE COMPLEX **for** mkl_zcoogemv.
Array, size at least m .
 On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  DOUBLE COMPLEX val(*), x(*), y(*)
```

C:

```
void mkl_scoogemv(char *transa, int *m, float *val, int *rowind,
int *colind, int *nnz, float *x, float *y);

void mkl_dcoogemv(char *transa, int *m, double *val, int *rowind,
int *colind, int *nnz, double *x, double *y);

void mkl_ccoogemv(char *transa, int *m, MKL_Complex8 *val, int *rowind,
int *colind, int *nnz, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zcoogemv(char *transa, int *m, MKL_Complex16 *val, int *rowind,
int *colind, int *nnz, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?diagemv

Computes matrix - vector product of a sparse general matrix stored in the diagonal format with one-based indexing.

Syntax**Fortran:**

```
call mkl_sdiagemv(transa, m, val, lval, iddiag, ndiag, x, y)
call mkl_ddiagemv(transa, m, val, lval, iddiag, ndiag, x, y)
call mkl_cdiagemv(transa, m, val, lval, iddiag, ndiag, x, y)
call mkl_zdiagemv(transa, m, val, lval, iddiag, ndiag, x, y)
```

C:

```
void mkl_sdiagemv (char *transa, MKL_INT *m, float *val, MKL_INT *lval, MKL_INT *iddiag,
MKL_INT *ndiag, float *x, float *y);
void mkl_ddiagemv (char *transa, MKL_INT *m, double *val, MKL_INT *lval, MKL_INT
*iddiag, MKL_INT *ndiag, double *x, double *y);
void mkl_cdiagemv (char *transa, MKL_INT *m, MKL_Complex8 *val, MKL_INT *lval, MKL_INT
*iddiag, MKL_INT *ndiag, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zdiagemv (char *transa, MKL_INT *m, MKL_Complex16 *val, MKL_INT *lval, MKL_INT
*iddiag, MKL_INT *ndiag, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The mkl_?diagemv routine performs a matrix-vector operation defined as

$y := A \cdot x$

or

$y := A' \cdot x,$

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the diagonal storage format, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := A*x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := A'*x$,
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>val</i>	REAL for mkl_sdiagemv. DOUBLE PRECISION for mkl_ddiagemv. COMPLEX for mkl_ccsrgemv. DOUBLE COMPLEX for mkl_zdiagemv. Two-dimensional array of size <i>lval</i> * <i>ndiag</i> , contains non-zero diagonals of the matrix A . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix A . Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix A .
<i>x</i>	REAL for mkl_sdiagemv. DOUBLE PRECISION for mkl_ddiagemv. COMPLEX for mkl_ccsrgemv. DOUBLE COMPLEX for mkl_zdiagemv. Array, size is <i>m</i> . On entry, the array <i>x</i> must contain the vector x .

Output Parameters

<i>y</i>	REAL for mkl_sdiagemv. DOUBLE PRECISION for mkl_ddiagemv. COMPLEX for mkl_ccsrgemv.
----------	---

DOUBLE COMPLEX for mkl_zdiagemv.
Array, size at least m .
 On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiagemv(transa, m, val, lval, iddiag, ndiag, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, lval, ndiag
```

```
  INTEGER iddiag(*)
```

```
  REAL val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_ddiagemv(transa, m, val, lval, iddiag, ndiag, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, lval, ndiag
```

```
  INTEGER iddiag(*)
```

```
  DOUBLE PRECISION val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_cdiagemv(transa, m, val, lval, iddiag, ndiag, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, lval, ndiag
```

```
  INTEGER iddiag(*)
```

```
  COMPLEX val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_zdiagemv(transa, m, val, lval, iddiag, ndiag, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, lval, ndiag
```

```
  INTEGER iddiag(*)
```

```
  DOUBLE COMPLEX val(lval,*), x(*), y(*)
```

C:

```
void mkl_sdiagemv(char *transa, int *m, float *val, int *lval,
int *iddiag, int *ndiag, float *x, float *y);

void mkl_ddiagemv(char *transa, int *m, double *val, int *lval,
int *iddiag, int *ndiag, double *x, double *y);

void mkl_cdiagemv(char *transa, int *m, MKL_Complex8 *val, int *lval,
int *iddiag, int *ndiag, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zdiagemv(char *transa, int *m, MKL_Complex16 *val, int *lval,
int *iddiag, int *ndiag, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?csrsymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the CSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_scsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_dcsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_ccsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_zcsrsymv(uplo, m, a, ia, ja, x, y)
```

C:

```
void mkl_scsrsymv (char *uplo, MKL_INT *m, float *a, MKL_INT *ia, MKL_INT *ja, float *x, float *y);
void mkl_dcsrsymv (char *uplo, MKL_INT *m, double *a, MKL_INT *ia, MKL_INT *ja, double *x, double *y);
void mkl_ccsrsymv (char *uplo, MKL_INT *m, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zcsrsymv (char *uplo, MKL_INT *m, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?csrsymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

`x` and `y` are vectors,

`A` is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (3-array variation).

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix <code>A</code> is used. If <code>uplo = 'U'</code> or <code>'u'</code> , then the upper triangle of the matrix <code>A</code> is used.
-------------------	--

If `uplo = 'L'` or `'l'`, then the low triangle of the matrix A is used.

`m`

INTEGER. Number of rows of the matrix A .

`a`

REAL **for** `mkl_scsrsymv`.

DOUBLE PRECISION **for** `mkl_dcsrsymv`.

COMPLEX **for** `mkl_ccsrsymv`.

DOUBLE COMPLEX **for** `mkl_zcsrsymv`.

Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to `values` array description in [Sparse Matrix Storage Formats](#) for more details.

`ia`

INTEGER. Array of length $m + 1$, containing indices of elements in the array `a`, such that `ia(i)` is the index in the array `a` of the first non-zero element from the row i . The value of the last element `ia(m + 1)` is equal to the number of non-zeros plus one. Refer to `rowIndex` array description in [Sparse Matrix Storage Formats](#) for more details.

`ja`

INTEGER. Array containing the column indices for each non-zero element of the matrix A .

Its length is equal to the length of the array `a`. Refer to `columns` array description in [Sparse Matrix Storage Formats](#) for more details.

`x`

REAL **for** `mkl_scsrsymv`.

DOUBLE PRECISION **for** `mkl_dcsrsymv`.

COMPLEX **for** `mkl_ccsrsymv`.

DOUBLE COMPLEX **for** `mkl_zcsrsymv`.

Array, size is m .

On entry, the array `x` must contain the vector x .

Output Parameters

`y`

REAL **for** `mkl_scsrsymv`.

DOUBLE PRECISION **for** `mkl_dcsrsymv`.

COMPLEX **for** `mkl_ccsrsymv`.

DOUBLE COMPLEX **for** `mkl_zcsrsymv`.

Array, size at least m .

On exit, the array `y` must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsymv(uplo, m, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m
  INTEGER ia(*), ja(*)
  REAL a(*), x(*), y(*)

SUBROUTINE mkl_dcsrsymv(uplo, m, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m
  INTEGER ia(*), ja(*)
  DOUBLE PRECISION a(*), x(*), y(*)

SUBROUTINE mkl_ccsrsymv(uplo, m, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m
  INTEGER ia(*), ja(*)
  COMPLEX a(*), x(*), y(*)

SUBROUTINE mkl_zcsrsymv(uplo, m, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m
  INTEGER ia(*), ja(*)
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_scsrsymv(char *uplo, int *m, float *a,
  int *ia, int *ja, float *x, float *y);

void mkl_dcsrsymv(char *uplo, int *m, double *a,
  int *ia, int *ja, double *x, double *y);

void mkl_ccsrsymv(char *uplo, int *m, MKL_Complex8 *a,
  int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zcsrsymv(char *uplo, int *m, MKL_Complex16 *a,
  int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?bsrsymv

Computes matrix-vector product of a sparse symmetrical matrix stored in the BSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_sbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_cbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_zbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

C:

```
void mkl_sbsrsymv (char *uplo, MKL_INT *m, MKL_INT *lb, float *a, MKL_INT *ia, MKL_INT *ja, float *x, float *y);
void mkl_dbsrsymv (char *uplo, MKL_INT *m, MKL_INT *lb, double *a, MKL_INT *ia, MKL_INT *ja, double *x, double *y);
void mkl_cbsrsymv (char *uplo, MKL_INT *m, MKL_INT *lb, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zbsrsymv (char *uplo, MKL_INT *m, MKL_INT *lb, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?bsrsymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation).

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is considered. If <code>uplo = 'U'</code> or ' <code>u</code> ', then the upper triangle of the matrix A is used.
-------------------	---

If $uplo = 'L'$ or $'l'$, then the low triangle of the matrix A is used.

m

INTEGER. Number of block rows of the matrix A .

lb

INTEGER. Size of the block in the matrix A .

a

REAL **for** mkl_sbsrsymv.

DOUBLE PRECISION **for** mkl_dbsrsymv.

COMPLEX **for** mkl_cbsrsymv.

DOUBLE COMPLEX **for** mkl_zcsrgemv.

Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb * lb$. Refer to *values* array description in [BSR Format](#) for more details.

ia

INTEGER. Array of length $(m + 1)$, containing indices of block in the array *a*, such that $ia(i)$ is the index in the array *a* of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in [BSR Format](#) for more details.

ja

INTEGER. Array containing the column indices for each non-zero block in the matrix A .

Its length is equal to the number of non-zero blocks of the matrix A . Refer to *columns* array description in [BSR Format](#) for more details.

x

REAL **for** mkl_sbsrsymv.

DOUBLE PRECISION **for** mkl_dbsrsymv.

COMPLEX **for** mkl_cbsrsymv.

DOUBLE COMPLEX **for** mkl_zcsrgemv.

Array, size $(m * lb)$.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y

REAL **for** mkl_sbsrsymv.

DOUBLE PRECISION **for** mkl_dbsrsymv.

COMPLEX **for** mkl_cbsrsymv.

DOUBLE COMPLEX **for** mkl_zcsrgemv.

Array, size at least $(m * lb)$.

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrsymv(uplo, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  REAL a(*), x(*), y(*)

SUBROUTINE mkl_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  DOUBLE PRECISION a(*), x(*), y(*)

SUBROUTINE mkl_cbsrsymv(uplo, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  COMPLEX a(*), x(*), y(*)

SUBROUTINE mkl_zbsrsymv(uplo, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_sbsrsymv(char *uplo, int *m, int *lb,
float *a, int *ia, int *ja, float *x, float *y);

void mkl_dbsrsymv(char *uplo, int *m, int *lb,
double *a, int *ia, int *ja, double *x, double *y);

void mkl_cbsrsymv(char *uplo, int *m, int *lb,
MKL_Complex8 *a, int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zbsrsymv(char *uplo, int *m, int *lb,
MKL_Complex16 *a, int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?coosymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with one-based indexing.

Syntax

Fortran:

```
call mkl_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_ccoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

C:

```
void mkl_scoosymv (char *uplo, MKL_INT *m, float *val, MKL_INT *rowind, MKL_INT
*colind, MKL_INT *nnz, float *x, float *y);

void mkl_dcoosymv (char *uplo, MKL_INT *m, double *val, MKL_INT *rowind, MKL_INT
*colind, MKL_INT *nnz, double *x, double *y);

void mkl_ccoosymv (char *uplo, MKL_INT *m, MKL_Complex8 *val, MKL_INT *rowind, MKL_INT
*colind, MKL_INT *nnz, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zcoosymv (char *uplo, MKL_INT *m, MKL_Complex16 *val, MKL_INT *rowind, MKL_INT
*colind, MKL_INT *nnz, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The mkl_?coosymv routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

uplo CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is used.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

If `uplo` = 'L' or 'l', then the low triangle of the matrix A is used.

`m` INTEGER. Number of rows of the matrix A .

`val` REAL for `mkl_scoosymv`.

DOUBLE PRECISION for `mkl_dcoosymv`.

COMPLEX for `mkl_ccoosymv`.

DOUBLE COMPLEX for `mkl_zcoosymv`.

Array of length `nnz`, contains non-zero elements of the matrix A in the arbitrary order.

Refer to `values` array description in [Coordinate Format](#) for more details.

`rowind` INTEGER. Array of length `nnz`, contains the row indices for each non-zero element of the matrix A .

Refer to `rows` array description in [Coordinate Format](#) for more details.

`colind` INTEGER. Array of length `nnz`, contains the column indices for each non-zero element of the matrix A . Refer to `columns` array description in [Coordinate Format](#) for more details.

`nnz` INTEGER. Specifies the number of non-zero element of the matrix A .

Refer to `nnz` description in [Coordinate Format](#) for more details.

`x` REAL for `mkl_scoosymv`.

DOUBLE PRECISION for `mkl_dcoosymv`.

COMPLEX for `mkl_ccoosymv`.

DOUBLE COMPLEX for `mkl_zcoosymv`.

Array, size is `m`.

On entry, the array `x` must contain the vector x .

Output Parameters

`y` REAL for `mkl_scoosymv`.

DOUBLE PRECISION for `mkl_dcoosymv`.

COMPLEX for `mkl_ccoosymv`.

DOUBLE COMPLEX for `mkl_zcoosymv`.

Array, size at least `m`.

On exit, the array `y` must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 uplo
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 uplo
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cdoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 uplo
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 uplo
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  DOUBLE COMPLEX val(*), x(*), y(*)
```

C:

```
void mkl_scoosymv(char *uplo, int *m, float *val, int *rowind,
int *colind, int *nnz, float *x, float *y);
```

```
void mkl_dcoosymv(char *uplo, int *m, double *val, int *rowind,
int *colind, int *nnz, double *x, double *y);
```

```
void mkl_ccoosymv(char *uplo, int *m, MKL_Complex8 *val, int *rowind,
int *colind, int *nnz, MKL_Complex8 *x, MKL_Complex8 *y);
```

```
void mkl_zcoosymv(char *uplo, int *m, MKL_Complex16 *val, int *rowind,
int *colind, int *nnz, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?diasymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the diagonal format with one-based indexing.

Syntax

Fortran:

```
call mkl_sdiasymv(uplo, m, val, lval, iddiag, ndiag, x, y)
call mkl_ddiasymv(uplo, m, val, lval, iddiag, ndiag, x, y)
call mkl_cdiasymv(uplo, m, val, lval, iddiag, ndiag, x, y)
call mkl_zdiasymv(uplo, m, val, lval, iddiag, ndiag, x, y)
```

C:

```
void mkl_sdiasymv (char *uplo, MKL_INT *m, float *val, MKL_INT *lval, MKL_INT *iddiag,
MKL_INT *ndiag, float *x, float *y);

void mkl_ddiasymv (char *uplo, MKL_INT *m, double *val, MKL_INT *lval, MKL_INT *iddiag,
MKL_INT *ndiag, double *x, double *y);

void mkl_cdiasymv (char *uplo, MKL_INT *m, MKL_Complex8 *val, MKL_INT *lval, MKL_INT
*iddiag, MKL_INT *ndiag, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zdiasymv (char *uplo, MKL_INT *m, MKL_Complex16 *val, MKL_INT *lval, MKL_INT
*iddiag, MKL_INT *ndiag, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The mkl_?diasymv routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

uplo CHARACTER*1. Specifies whether the upper or low triangle of the matrix *A* is used.

If *uplo* = 'U' or 'u', then the upper triangle of the matrix *A* is used.

If `uplo` = 'L' or 'l', then the low triangle of the matrix A is used.

`m` INTEGER. Number of rows of the matrix A .

`val` REAL for `mkl_sdiasymv`.

DOUBLE PRECISION for `mkl_ddiasymv`.

COMPLEX for `mkl_cdiasymv`.

DOUBLE COMPLEX for `mkl_zdiasymv`.

Two-dimensional array of size `lval` by `ndiag`, contains non-zero diagonals of the matrix A . Refer to `values` array description in [Diagonal Storage Scheme](#) for more details.

`lval` INTEGER. Leading dimension of `val`, $lval \geq m$. Refer to `lval` description in [Diagonal Storage Scheme](#) for more details.

`idiag` INTEGER. Array of length `ndiag`, contains the distances between main diagonal and each non-zero diagonals in the matrix A .

Refer to `distance` array description in [Diagonal Storage Scheme](#) for more details.

`ndiag` INTEGER. Specifies the number of non-zero diagonals of the matrix A .

`x` REAL for `mkl_sdiasymv`.

DOUBLE PRECISION for `mkl_ddiasymv`.

COMPLEX for `mkl_cdiasymv`.

DOUBLE COMPLEX for `mkl_zdiasymv`.

Array, size is m .

On entry, the array `x` must contain the vector x .

Output Parameters

`y` REAL for `mkl_sdiasymv`.

DOUBLE PRECISION for `mkl_ddiasymv`.

COMPLEX for `mkl_cdiasymv`.

DOUBLE COMPLEX for `mkl_zdiasymv`.

Array, size at least m .

On exit, the array `y` must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiasymv(uplo, m, val, lval, iddiag, ndiag, x, y)
  CHARACTER*1    uplo
  INTEGER        m, lval, ndiag
  INTEGER        iddiag(*)
  REAL           val(lval,*), x(*), y(*)

SUBROUTINE mkl_ddiasymv(uplo, m, val, lval, iddiag, ndiag, x, y)
  CHARACTER*1    uplo
  INTEGER        m, lval, ndiag
  INTEGER        iddiag(*)
  DOUBLE PRECISION    val(lval,*), x(*), y(*)

SUBROUTINE mkl_cdiasymv(uplo, m, val, lval, iddiag, ndiag, x, y)
  CHARACTER*1    uplo
  INTEGER        m, lval, ndiag
  INTEGER        iddiag(*)
  COMPLEX        val(lval,*), x(*), y(*)

SUBROUTINE mkl_zdiasymv(uplo, m, val, lval, iddiag, ndiag, x, y)
  CHARACTER*1    uplo
  INTEGER        m, lval, ndiag
  INTEGER        iddiag(*)
  DOUBLE COMPLEX    val(lval,*), x(*), y(*)
```

C:

```
void mkl_sdiasymv(char *uplo, int *m, float *val, int *lval,
int *idiag, int *ndiag, float *x, float *y);

void mkl_ddiasymv(char *uplo, int *m, double *val, int *lval,
int *idiag, int *ndiag, double *x, double *y);

void mkl_cdiasymv(char *uplo, int *m, MKL_Complex8 *val, int *lval,
int *idiag, int *ndiag, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zdiasymv(char *uplo, int *m, MKL_Complex16 *val, int *lval,
int *idiag, int *ndiag, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?csrtrs

Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_scsrtrs(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_dcsrtrs(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_ccsrtrs(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_zcsrtrs(uplo, transa, diag, m, a, ia, ja, x, y)
```

C:

```
void mkl_scsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, float *a, MKL_INT
*ia, MKL_INT *ja, float *x, float *y);

void mkl_dcsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, double *a,
MKL_INT *ia, MKL_INT *ja, double *x, double *y);

void mkl_ccsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_Complex8 *a,
MKL_INT *ia, MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zcsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_Complex16 *a,
MKL_INT *ia, MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?csrtrs` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3 array variation):

$A^*y = x$

or

$A'^*y = x,$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used. If $uplo = 'U'$ or ' u ', then the upper triangle of the matrix A is used. If $uplo = 'L'$ or ' l ', then the low triangle of the matrix A is used.
<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If $transa = 'N'$ or ' n ', then $A^*y = x$ If $transa = 'T'$ or ' t ' or ' C ' or ' c ', then $A'^*y = x$,
<i>diag</i>	CHARACTER*1. Specifies whether A is unit triangular. If $diag = 'U'$ or ' u ', then A is a unit triangular. If $diag = 'N'$ or ' n ', then A is not unit triangular.
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>a</i>	REAL for mkl_scsrtrmv. DOUBLE PRECISION for mkl_dcsrtrmv. COMPLEX for mkl_ccsrtrmv. DOUBLE COMPLEX for mkl_zcsrtrmv. Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>ia</i>	INTEGER. Array of length $m + 1$, containing indices of elements in the array <i>a</i> , such that $ia(i)$ is the index in the array <i>a</i> of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
-----------	---

<i>ja</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix A . Its length is equal to the length of the array <i>a</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
-----------	--

NOTE

Column indices must be sorted in increasing order for each row.

<i>x</i>	REAL for mkl_scsrtrmv. DOUBLE PRECISION for mkl_dcsrtrmv.
----------	--

COMPLEX for mkl_ccsrtrmv.
 DOUBLE COMPLEX for mkl_zcsrtrmv.
Array, size is m .
 On entry, the array x must contain the vector x .

Output Parameters

y
 REAL for mkl_scsrtrmv.
 DOUBLE PRECISION for mkl_dcsrtrmv.
 COMPLEX for mkl_ccsrtrmv.
 DOUBLE COMPLEX for mkl_zcsrtrmv.
Array, size at least m .
 Contains the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m
```

```
  INTEGER ia(*), ja(*)
```

```
  REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m
```

```
  INTEGER ia(*), ja(*)
```

```
  DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m
```

```
  INTEGER ia(*), ja(*)
```

```
  COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m
```

```
  INTEGER ia(*), ja(*)
```

```
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_scsrtrs(char *uplo, char *transa, char *diag, int *m,
float *a, int *ia, int *ja, float *x, float *y);

void mkl_dcsrtrs(char *uplo, char *transa, char *diag, int *m,
double *a, int *ia, int *ja, double *x, double *y);

void mkl_ccsrtrs(char *uplo, char *transa, char *diag, int *m,
MKL_Complex8 *a, int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zcsrtrs(char *uplo, char *transa, char *diag, int *m,
MKL_Complex16 *a, int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?bsrtrs

Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with one-based indexing.

Syntax**Fortran:**

```
call mkl_sbsrtrs(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_dbsrtrs(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_cbsrtrs(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_zbsrtrs(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

C:

```
void mkl_sbsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_INT *lb,
float *a, MKL_INT *ia, MKL_INT *ja, float *x, float *y);

void mkl_dbsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_INT *lb,
double *a, MKL_INT *ia, MKL_INT *ja, double *x, double *y);

void mkl_cbsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_INT *lb,
MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zbsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_INT *lb,
MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The mkl_?bsrtrs routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation) :

$y := A \cdot x$

or

$y := A' \cdot x,$

where:

x and *y* are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	CHARACTER*1. Specifies the upper or low triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.
<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as <i>y</i> := <i>A</i> * <i>x</i> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as <i>y</i> := <i>A'</i> * <i>x</i> .
<i>diag</i>	CHARACTER*1. Specifies whether <i>A</i> is a unit triangular matrix. If <i>diag</i> = 'U' or 'u', then <i>A</i> is a unit triangular. If <i>diag</i> = 'N' or 'n', then <i>A</i> is not a unit triangular.
<i>m</i>	INTEGER. Number of block rows of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>a</i>	REAL for mkl_sbsrtrsv. DOUBLE PRECISION for mkl_dbsrtrsv. COMPLEX for mkl_cbsrtrsv. DOUBLE COMPLEX for mkl_zbsrtrsv. Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb</i> * <i>lb</i> . Refer to <i>values</i> array description in BSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>ia</i>	INTEGER. Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that $ia(I)$ is the index in the array <i>a</i> of the first non-zero element from the row I . The value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks plus one. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks of the matrix <i>A</i> . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	REAL for mkl_sbsrtrs v. DOUBLE PRECISION for mkl_dbsrtrs v. COMPLEX for mkl_cbsrtrs v. DOUBLE COMPLEX for mkl_zbsrtrs v. Array, size $(m * lb)$. On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL for mkl_sbsrtrs v. DOUBLE PRECISION for mkl_dbsrtrs v. COMPLEX for mkl_cbsrtrs v. DOUBLE COMPLEX for mkl_zbsrtrs v. Array, size at least $(m * lb)$. On exit, the array <i>y</i> must contain the vector <i>y</i> .
----------	--

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrtrs v(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
    CHARACTER*1 uplo, transa, diag
```

```
    INTEGER m, lb
```

```
    INTEGER ia(*), ja(*)
```

```
    REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_dbsrtrs v(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

```
    CHARACTER*1 uplo, transa, diag
```

```
    INTEGER m, lb
```

```
    INTEGER ia(*), ja(*)
```

```
    DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_zbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_sbsrtrsv(char *uplo, char *transa, char *diag, int *m,
int *lb, float *a, int *ia, int *ja, float *x, float *y);
void mkl_dbsrtrsv(char *uplo, char *transa, char *diag, int *m,
int *lb, double *a, int *ia, int *ja, double *x, double *y);
void mkl_cbsrtrsv(char *uplo, char *transa, char *diag, int *m,
int *lb, MKL_Complex8 *a, int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zbsrtrsv(char *uplo, char *transa, char *diag, int *m,
int *lb, MKL_Complex16 *a, int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?cootrv

Triangular solvers with simplified interface for a sparse matrix in the coordinate format with one-based indexing.

Syntax**Fortran:**

```
call mkl_scootrv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_dcootrv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_ccootrv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_zcootrv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

C:

```
void mkl_scootrv (char *uplo, char *transa, char *diag, MKL_INT *m, float *val,
MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, float *x, float *y);
void mkl_dcootrv (char *uplo, char *transa, char *diag, MKL_INT *m, double *val,
MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, double *x, double *y);
void mkl_ccootrv (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_Complex8 *val,
MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, MKL_Complex8 *x, MKL_Complex8 *y);
```

```
void mkl_zcootrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_Complex16
*val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, MKL_Complex16 *x, MKL_Complex16
*y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?cootrs` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format:

$A^*y = x$

or

$A'^*y = x$,

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is considered. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A^*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A'^*y = x$,
<i>diag</i>	CHARACTER*1. Specifies whether A is unit triangular. If <i>diag</i> = 'U' or 'u', then A is unit triangular. If <i>diag</i> = 'N' or 'n', then A is not unit triangular.
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>val</i>	REAL for <code>mkl_scootrs</code> . DOUBLE PRECISION for <code>mkl_dcootrs</code> . COMPLEX for <code>mkl_ccootrs</code> . DOUBLE COMPLEX for <code>mkl_zcootrs</code> .

Array of length nnz , contains non-zero elements of the matrix A in the arbitrary order.

Refer to $values$ array description in [Coordinate Format](#) for more details.

$rowind$

INTEGER. Array of length nnz , contains the row indices for each non-zero element of the matrix A .

Refer to $rows$ array description in [Coordinate Format](#) for more details.

$colind$

INTEGER. Array of length nnz , contains the column indices for each non-zero element of the matrix A . Refer to $columns$ array description in [Coordinate Format](#) for more details.

nnz

INTEGER. Specifies the number of non-zero element of the matrix A .

Refer to nnz description in [Coordinate Format](#) for more details.

x

REAL **for** mkl_scootrs.

DOUBLE PRECISION **for** mkl_dcootrs.

COMPLEX **for** mkl_ccootrs.

DOUBLE COMPLEX **for** mkl_zcootrs.

Array, size is m .

On entry, the array x must contain the vector x .

Output Parameters

y

REAL **for** mkl_scootrs.

DOUBLE PRECISION **for** mkl_dcootrs.

COMPLEX **for** mkl_ccootrs.

DOUBLE COMPLEX **for** mkl_zcootrs.

Array, size at least m .

Contains the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scootrs(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
CHARACTER*1 uplo, transa, diag
INTEGER m, nnz
INTEGER rowind(*), colind(*)
REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, nnz
  INTEGER rowind(*), colind(*)
  DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, nnz
  INTEGER rowind(*), colind(*)
  COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcootrsv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, nnz
  INTEGER rowind(*), colind(*)
  DOUBLE COMPLEX val(*), x(*), y(*)
```

C:

```
void mkl_scootrsv(char *uplo, char *transa, char *diag, int *m,
float *val, int *rowind, int *colind, int *nnz, float *x, double *y);
void mkl_dcootrsv(char *uplo, char *transa, char *diag, int *m,
double *val, int *rowind, int *colind, int *nnz, double *x, double *y);
void mkl_ccootrsv(char *uplo, char *transa, char *diag, int *m,
MKL_Complex8 *val, int *rowind, int *colind, int *nnz, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zcootrsv(char *uplo, char *transa, char *diag, int *m,
MKL_Complex16 *val, int *rowind, int *colind, int *nnz, MKL_Complex16 *x, MKL_Complex16 *y);
```

[mkl_?diatrs](#)

Triangular solvers with simplified interface for a sparse matrix in the diagonal format with one-based indexing.

Syntax

Fortran:

```
call mkl_sdiatrsv(uplo, transa, diag, m, val, lval, iddiag, ndiag, x, y)
call mkl_ddiatrsv(uplo, transa, diag, m, val, lval, iddiag, ndiag, x, y)
call mkl_cdiatrsv(uplo, transa, diag, m, val, lval, iddiag, ndiag, x, y)
call mkl_zdiatrsv(uplo, transa, diag, m, val, lval, iddiag, ndiag, x, y)
```

C:

```
void mkl_sdiatrs (char *uplo, char *transa, char *diag, MKL_INT *m, float *val,
MKL_INT *lval, MKL_INT *idiag, MKL_INT *ndiag, float *x, float *y);
void mkl_ddiatrs (char *uplo, char *transa, char *diag, MKL_INT *m, double *val,
MKL_INT *lval, MKL_INT *idiag, MKL_INT *ndiag, double *x, double *y);
void mkl_cdiatrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_Complex8 *val,
MKL_INT *lval, MKL_INT *idiag, MKL_INT *ndiag, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zdiatrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_Complex16
*val, MKL_INT *lval, MKL_INT *idiag, MKL_INT *ndiag, MKL_Complex16 *x, MKL_Complex16
*y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?diatrs` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

$A^*y = x$

or

$A'^*y = x,$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used. If <code>uplo</code> = 'U' or 'u', then the upper triangle of the matrix A is used. If <code>uplo</code> = 'L' or 'l', then the low triangle of the matrix A is used.
<code>transa</code>	CHARACTER*1. Specifies the system of linear equations. If <code>transa</code> = 'N' or 'n', then $A^*y = x$ If <code>transa</code> = 'T' or 't' or 'C' or 'c', then $A'^*y = x$,
<code>diag</code>	CHARACTER*1. Specifies whether A is unit triangular. If <code>diag</code> = 'U' or 'u', then A is unit triangular. If <code>diag</code> = 'N' or 'n', then A is not unit triangular.

<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>val</i>	<p>REAL for mkl_sdiatrv.</p> <p>DOUBLE PRECISION for mkl_ddiatrv.</p> <p>COMPLEX for mkl_cdiatrv.</p> <p>DOUBLE COMPLEX for mkl_zdiatrv.</p>
	Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , $lval \geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> .

NOTE

All elements of this array must be sorted in increasing order.

	Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>x</i>	<p>REAL for mkl_sdiatrv.</p> <p>DOUBLE PRECISION for mkl_ddiatrv.</p> <p>COMPLEX for mkl_cdiatrv.</p> <p>DOUBLE COMPLEX for mkl_zdiatrv.</p>
	Array, size is <i>m</i> .
	On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	<p>REAL for mkl_sdiatrv.</p> <p>DOUBLE PRECISION for mkl_ddiatrv.</p> <p>COMPLEX for mkl_cdiatrv.</p> <p>DOUBLE COMPLEX for mkl_zdiatrv.</p>
	Array, size at least <i>m</i> .

Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiatrsv(uplo, transa, diag, m, val, lval, iddiag, ndiag, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m, lval, ndiag
```

```
  INTEGER iddiag(*)
```

```
  REAL val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_ddiatrsv(uplo, transa, diag, m, val, lval, iddiag, ndiag, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m, lval, ndiag
```

```
  INTEGER iddiag(*)
```

```
  DOUBLE PRECISION val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_cdiatrsv(uplo, transa, diag, m, val, lval, iddiag, ndiag, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m, lval, ndiag
```

```
  INTEGER iddiag(*)
```

```
  COMPLEX val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_zdiatrsv(uplo, transa, diag, m, val, lval, iddiag, ndiag, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m, lval, ndiag
```

```
  INTEGER iddiag(*)
```

```
  DOUBLE COMPLEX val(lval,*), x(*), y(*)
```

C:

```
void mkl_sdiatrsv(char *uplo, char *transa, char *diag, int *m, float
```

```
*val, int *lval, int *idiag, int *ndiag, float *x, float *y);
```

```
void mkl_ddiatrsv(char *uplo, char *transa, char *diag, int *m, double
```

```
*val, int *lval, int *idiag, int *ndiag, double *x, double *y);
```

```
void mkl_cdiatrsv(char *uplo, char *transa, char *diag, int *m, MKL_Complex8
```

```
*val, int *lval, int *idiag, int *ndiag, MKL_Complex8 *x, MKL_Complex8 *y);
```

```
void mkl_zdiatrsv(char *uplo, char *transa, char *diag, int *m, MKL_Complex16
```

```
*val, int *lval, int *idiag, int *ndiag, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_cspblas_?csrgemv

Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_scsrgemv(transa, m, a, ia, ja, x, y)
call mkl_cspblas_dcsrgemv(transa, m, a, ia, ja, x, y)
call mkl_cspblas_ccsrgemv(transa, m, a, ia, ja, x, y)
call mkl_cspblas_zcsrgemv(transa, m, a, ia, ja, x, y)
```

C:

```
void mkl_cspblas_scsrgemv (char *transa, MKL_INT *m, float *a, MKL_INT *ia, MKL_INT
*ja, float *x, float *y);

void mkl_cspblas_dcsrgemv (char *transa, MKL_INT *m, double *a, MKL_INT *ia, MKL_INT
*ja, double *x, double *y);

void mkl_cspblas_ccsrgemv (char *transa, MKL_INT *m, MKL_Complex8 *a, MKL_INT *ia,
MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zcsrgemv (char *transa, MKL_INT *m, MKL_Complex16 *a, MKL_INT *ia,
MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_cspblas_?csrgemv` routine performs a matrix-vector operation defined as

$y := A^*x$

or

$y := A'^*x,$

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the CSR format (3-array variation) with zero-based indexing, A' is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

`transa` CHARACTER*1. Specifies the operation.

If $\text{transa} = \text{'N'}$ or 'n' , then the matrix-vector product is computed as
 $y := A^*x$

If $\text{transa} = \text{'T'}$ or 't' or 'C' or 'c' , then the matrix-vector product is computed as $y := A'^*x$,

m

INTEGER. Number of rows of the matrix A .

a

REAL **for** mkl_cspblas_scsrgemv.

DOUBLE PRECISION **for** mkl_cspblas_dcsrgemv.

COMPLEX **for** mkl_cspblas_ccsrgemv.

DOUBLE COMPLEX **for** mkl_cspblas_zcsrgemv.

Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ia

INTEGER. Array of length $m + 1$, containing indices of elements in the array *a*, such that $\text{ia}(I)$ is the index in the array *a* of the first non-zero element from the row I . The value of the last element $\text{ia}(m)$ is equal to the number of non-zeros. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

ja

INTEGER. Array containing the column indices for each non-zero element of the matrix A .

Its length is equal to the length of the array *a*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

x

REAL **for** mkl_cspblas_scsrgemv.

DOUBLE PRECISION **for** mkl_cspblas_dcsrgemv.

COMPLEX **for** mkl_cspblas_ccsrgemv.

DOUBLE COMPLEX **for** mkl_cspblas_zcsrgemv.

Array, size is *m*.

One entry, the array *x* must contain the vector *x*.

Output Parameters

y

REAL **for** mkl_cspblas_scsrgemv.

DOUBLE PRECISION **for** mkl_cspblas_dcsrgemv.

COMPLEX **for** mkl_cspblas_ccsrgemv.

DOUBLE COMPLEX **for** mkl_cspblas_zcsrgemv.

Array, size at least *m*.

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scsrgemv(transa, m, a, ia, ja, x, y)
  CHARACTER*1 transa
  INTEGER m
  INTEGER ia(*), ja(*)
  REAL a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dcsrgemv(transa, m, a, ia, ja, x, y)
  CHARACTER*1 transa
  INTEGER m
  INTEGER ia(*), ja(*)
  DOUBLE PRECISION a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_ccsrgemv(transa, m, a, ia, ja, x, y)
  CHARACTER*1 transa
  INTEGER m
  INTEGER ia(*), ja(*)
  COMPLEX a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zcsrgemv(transa, m, a, ia, ja, x, y)
  CHARACTER*1 transa
  INTEGER m
  INTEGER ia(*), ja(*)
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_scsrgemv(char *transa, int *m, float *a,
                           int *ia, int *ja, float *x, float *y);

void mkl_cspblas_dcsrgemv(char *transa, int *m, double *a,
                           int *ia, int *ja, double *x, double *y);

void mkl_cspblas_ccsrgemv(char *transa, int *m, MKL_Complex8 *a,
                           int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zcsrgemv(char *transa, int *m, MKL_Complex16 *a,
                           int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_cspblas_?bsrgemv

Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cspblas_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cspblas_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
call mkl_cspblas_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
```

C:

```
void mkl_cspblas_sbsrgemv (char *transa, MKL_INT *m, MKL_INT *lb, float *a, MKL_INT
*ia, MKL_INT *ja, float *x, float *y);
void mkl_cspblas_dbsrgemv (char *transa, MKL_INT *m, MKL_INT *lb, double *a, MKL_INT
*ia, MKL_INT *ja, double *x, double *y);
void mkl_cspblas_cbsrgemv (char *transa, MKL_INT *m, MKL_INT *lb, MKL_Complex8 *a,
MKL_INT *ia, MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_cspblas_zbsrgemv (char *transa, MKL_INT *m, MKL_INT *lb, MKL_Complex16 *a,
MKL_INT *ia, MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_cspblas_?bsrgemv` routine performs a matrix-vector operation defined as

$y := A^*x$

or

$y := A'^*x,$

where:

x and y are vectors,

A is an m -by- m block sparse square matrix in the BSR format (3-array variation) with zero-based indexing, A' is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

`transa` CHARACTER*1. Specifies the operation.

If $\text{transa} = \text{'N'}$ or 'n' , then the matrix-vector product is computed as
 $y := A^*x$

If $\text{transa} = \text{'T'}$ or 't' or 'C' or 'c' , then the matrix-vector product is computed as $y := A'^*x$,

m

INTEGER. Number of block rows of the matrix A .

lb

INTEGER. Size of the block in the matrix A .

a

REAL **for** mkl_cspblas_sbsrgemv.

DOUBLE PRECISION **for** mkl_cspblas_dbsrgemv.

COMPLEX **for** mkl_cspblas_cbsrgemv.

DOUBLE COMPLEX **for** mkl_cspblas_zbsrgemv.

Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb * lb$. Refer to *values* array description in [BSR Format](#) for more details.

ia

INTEGER. Array of length $(m + 1)$, containing indices of block in the array *a*, such that $ia(i)$ is the index in the array *a* of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks. Refer to *rowIndex* array description in [BSR Format](#) for more details.

ja

INTEGER. Array containing the column indices for each non-zero block in the matrix A .

Its length is equal to the number of non-zero blocks of the matrix A . Refer to *columns* array description in [BSR Format](#) for more details.

x

REAL **for** mkl_cspblas_sbsrgemv.

DOUBLE PRECISION **for** mkl_cspblas_dbsrgemv.

COMPLEX **for** mkl_cspblas_cbsrgemv.

DOUBLE COMPLEX **for** mkl_cspblas_zbsrgemv.

Array, size $(m * lb)$.

On entry, the array *x* must contain the vector *x*.

Output Parameters

y

REAL **for** mkl_cspblas_sbsrgemv.

DOUBLE PRECISION **for** mkl_cspblas_dbsrgemv.

COMPLEX **for** mkl_cspblas_cbsrgemv.

DOUBLE COMPLEX **for** mkl_cspblas_zbsrgemv.

Array, size at least $(m * lb)$.

On exit, the array *y* must contain the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_sbsrgemv(transa, m, lb, a, ia, ja, x, y)
  CHARACTER*1 transa
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  REAL a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dbsrgemv(transa, m, lb, a, ia, ja, x, y)
  CHARACTER*1 transa
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  DOUBLE PRECISION a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_cbsrgemv(transa, m, lb, a, ia, ja, x, y)
  CHARACTER*1 transa
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  COMPLEX a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zbsrgemv(transa, m, lb, a, ia, ja, x, y)
  CHARACTER*1 transa
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_sbsrgemv(char *transa, int *m, int *lb,
                           float *a, int *ia, int *ja, float *x, float *y);

void mkl_cspblas_dbsrgemv(char *transa, int *m, int *lb,
                           double *a, int *ia, int *ja, double *x, double *y);

void mkl_cspblas_cbsrgemv(char *transa, int *m, int *lb,
                           MKL_Complex8 *a, int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zbsrgemv(char *transa, int *m, int *lb,
                           MKL_Complex16 *a, int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_cspblas_?coogemv

Computes matrix - vector product of a sparse general matrix stored in the coordinate format with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

C:

```
void mkl_cspblas_scoogemv (char *transa, MKL_INT *m, float *val, MKL_INT *rowind,
                            MKL_INT *colind, MKL_INT *nnz, float *x, float *y);
void mkl_cspblas_dcoogemv (char *transa, MKL_INT *m, double *val, MKL_INT *rowind,
                           MKL_INT *colind, MKL_INT *nnz, double *x, double *y);
void mkl_cspblas_ccoogemv (char *transa, MKL_INT *m, MKL_Complex8 *val, MKL_INT
                           *rowind, MKL_INT *colind, MKL_INT *nnz, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_cspblas_zcoogemv (char *transa, MKL_INT *m, MKL_Complex16 *val, MKL_INT
                           *rowind, MKL_INT *colind, MKL_INT *nnz, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_cspblas_dcoogemv` routine performs a matrix-vector operation defined as

$y := A^*x$

or

$y := A'^*x,$

where:

x and y are vectors,

A is an m -by- m sparse square matrix in the coordinate format with zero-based indexing, A' is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>transa</code>	CHARACTER*1. Specifies the operation.
---------------------	---------------------------------------

If $\text{transa} = \text{'N'}$ or 'n' , then the matrix-vector product is computed as

$y := A^*x$

If $\text{transa} = \text{'T'}$ or 't' or 'C' or 'c' , then the matrix-vector product is computed as $y := A'^*x$.

m

INTEGER. Number of rows of the matrix A .

val

REAL **for** mkl_cspblas_scoogemv.

DOUBLE PRECISION **for** mkl_cspblas_dcoogemv.

COMPLEX **for** mkl_cspblas_ccoogemv.

DOUBLE COMPLEX **for** mkl_cspblas_zcoogemv.

Array of length nnz , contains non-zero elements of the matrix A in the arbitrary order.

Refer to *values* array description in [Coordinate Format](#) for more details.

rowind

INTEGER. Array of length nnz , contains the row indices for each non-zero element of the matrix A .

Refer to *rows* array description in [Coordinate Format](#) for more details.

colind

INTEGER. Array of length nnz , contains the column indices for each non-zero element of the matrix A . Refer to *columns* array description in [Coordinate Format](#) for more details.

nnz

INTEGER. Specifies the number of non-zero element of the matrix A .

Refer to *nnz* description in [Coordinate Format](#) for more details.

x

REAL **for** mkl_cspblas_scoogemv.

DOUBLE PRECISION **for** mkl_cspblas_dcoogemv.

COMPLEX **for** mkl_cspblas_ccoogemv.

DOUBLE COMPLEX **for** mkl_cspblas_zcoogemv.

Array, size is m .

On entry, the array *x* must contain the vector x .

Output Parameters

y

REAL **for** mkl_cspblas_scoogemv.

DOUBLE PRECISION **for** mkl_cspblas_dcoogemv.

COMPLEX **for** mkl_cspblas_ccoogemv.

DOUBLE COMPLEX **for** mkl_cspblas_zcoogemv.

Array, size at least m .

On exit, the array *y* must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_ccoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zcoogemv(transa, m, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1 transa
```

```
  INTEGER m, nnz
```

```
  INTEGER rowind(*), colind(*)
```

```
  DOUBLE COMPLEX val(*), x(*), y(*)
```

C:

```
void mkl_cspblas_scoogemv(char *transa, int *m, float *val, int *rowind,
int *colind, int *nnz, float *x, float *y);
```

```
void mkl_cspblas_dcoogemv(char *transa, int *m, double *val, int *rowind,
int *colind, int *nnz, double *x, double *y);
```

```
void mkl_cspblas_ccoogemv(char *transa, int *m, MKL_Complex8 *val, int *rowind,
int *colind, int *nnz, MKL_Complex8 *x, MKL_Complex8 *y);
```

```
void mkl_cspblas_zcoogemv(char *transa, int *m, MKL_Complex16 *val, int *rowind,
int *colind, int *nnz, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_cspblas_?csrsymv

Computes matrix-vector product of a sparse symmetrical matrix stored in the CSR format (3-array variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_scsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_cspblas_dcsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_cspblas_ccsrsymv(uplo, m, a, ia, ja, x, y)
call mkl_cspblas_zcsrsymv(uplo, m, a, ia, ja, x, y)
```

C:

```
void mkl_cspblas_scsrsymv (char *uplo, MKL_INT *m, float *a, MKL_INT *ia, MKL_INT *ja,
float *x, float *y);

void mkl_cspblas_dcsrsymv (char *uplo, MKL_INT *m, double *a, MKL_INT *ia, MKL_INT *ja,
double *x, double *y);

void mkl_cspblas_ccsrsymv (char *uplo, MKL_INT *m, MKL_Complex8 *a, MKL_INT *ia,
MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zcsrsymv (char *uplo, MKL_INT *m, MKL_Complex16 *a, MKL_INT *ia,
MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_cspblas_?csrsymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (3-array variation) with zero-based indexing.

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used.
-------------------	---

If $uplo = 'U'$ or ' u ', then the upper triangle of the matrix A is used.

If $uplo = 'L'$ or ' l ', then the low triangle of the matrix A is used.

m

INTEGER. Number of rows of the matrix A .

a

REAL for `mkl_cspblas_scsrsymv`.

DOUBLE PRECISION for `mkl_cspblas_dcsrsymv`.

COMPLEX for `mkl_cspblas_ccsrsymv`.

DOUBLE COMPLEX for `mkl_cspblas_zcsrsymv`.

Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to `values` array description in [Sparse Matrix Storage Formats](#) for more details.

ia

INTEGER. Array of length $m + 1$, containing indices of elements in the array a , such that $ia(i)$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zeros. Refer to `rowIndex` array description in [Sparse Matrix Storage Formats](#) for more details.

ja

INTEGER. Array containing the column indices for each non-zero element of the matrix A .

Its length is equal to the length of the array a . Refer to `columns` array description in [Sparse Matrix Storage Formats](#) for more details.

x

REAL for `mkl_cspblas_scsrsymv`.

DOUBLE PRECISION for `mkl_cspblas_dcsrsymv`.

COMPLEX for `mkl_cspblas_ccsrsymv`.

DOUBLE COMPLEX for `mkl_cspblas_zcsrsymv`.

Array, size is m .

On entry, the array x must contain the vector x .

Output Parameters

y

REAL for `mkl_cspblas_scsrsymv`.

DOUBLE PRECISION for `mkl_cspblas_dcsrsymv`.

COMPLEX for `mkl_cspblas_ccsrsymv`.

DOUBLE COMPLEX for `mkl_cspblas_zcsrsymv`.

Array, size at least m .

On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scsrsymv(uplo, m, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m
  INTEGER ia(*), ja(*)
  REAL a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dcsrsymv(uplo, m, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m
  INTEGER ia(*), ja(*)
  DOUBLE PRECISION a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_ccsrsymv(uplo, m, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m
  INTEGER ia(*), ja(*)
  COMPLEX a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zcsrsymv(uplo, m, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m
  INTEGER ia(*), ja(*)
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_scsrsymv(char *uplo, int *m, float *a,
                           int *ia, int *ja, float *x, float *y);

void mkl_cspblas_dcsrsymv(char *uplo, int *m, double *a,
                           int *ia, int *ja, double *x, double *y);

void mkl_cspblas_ccsrsymv(char *uplo, int *m, MKL_Complex8 *a,
                           int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zcsrsymv(char *uplo, int *m, MKL_Complex16 *a,
                           int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_cspblas_?bsrsymv

Computes matrix-vector product of a sparse symmetrical matrix stored in the BSR format (3-arrays variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_sbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_cspblas_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_cspblas_cbsrsymv(uplo, m, lb, a, ia, ja, x, y)
call mkl_cspblas_zbsrsymv(uplo, m, lb, a, ia, ja, x, y)
```

C:

```
void mkl_cspblas_sbsrsymv (char *uplo, MKL_INT *m, MKL_INT *lb, float *a, MKL_INT *ia,
MKL_INT *ja, float *x, float *y);
void mkl_cspblas_dbsrsymv (char *uplo, MKL_INT *m, MKL_INT *lb, double *a, MKL_INT *ia,
MKL_INT *ja, double *x, double *y);
void mkl_cspblas_cbsrsymv (char *uplo, MKL_INT *m, MKL_INT *lb, MKL_Complex8 *a,
MKL_INT *ia, MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_cspblas_zbsrsymv (char *uplo, MKL_INT *m, MKL_INT *lb, MKL_Complex16 *a,
MKL_INT *ia, MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_cspblas_?bsrsymv` routine performs a matrix-vector operation defined as

$y := A^*x$

where:

x and y are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation) with zero-based indexing.

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is used.
-------------------	---

If $uplo = 'U'$ or ' u ', then the upper triangle of the matrix A is used.

If $uplo = 'L'$ or ' l ', then the low triangle of the matrix A is used.

m

INTEGER. Number of block rows of the matrix A .

lb

INTEGER. Size of the block in the matrix A .

a

REAL **for** mkl_cspblas_sbsrsymv.

DOUBLE PRECISION **for** mkl_cspblas_dbsrsymv.

COMPLEX **for** mkl_cspblas_cbsrsymv.

DOUBLE COMPLEX **for** mkl_cspblas_zbsrsymv.

Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $lb * lb$. Refer to *values* array description in [BSR Format](#) for more details.

ia

INTEGER. Array of length $(m + 1)$, containing indices of block in the array a , such that $ia(i)$ is the index in the array a of the first non-zero element from the row i . The value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks plus one. Refer to *rowIndex* array description in [BSR Format](#) for more details.

ja

INTEGER. Array containing the column indices for each non-zero block in the matrix A .

Its length is equal to the number of non-zero blocks of the matrix A . Refer to *columns* array description in [BSR Format](#) for more details.

x

REAL **for** mkl_cspblas_sbsrsymv.

DOUBLE PRECISION **for** mkl_cspblas_dbsrsymv.

COMPLEX **for** mkl_cspblas_cbsrsymv.

DOUBLE COMPLEX **for** mkl_cspblas_zbsrsymv.

Array, size $(m * lb)$.

On entry, the array x must contain the vector x .

Output Parameters

y

REAL **for** mkl_cspblas_sbsrsymv.

DOUBLE PRECISION **for** mkl_cspblas_dbsrsymv.

COMPLEX **for** mkl_cspblas_cbsrsymv.

DOUBLE COMPLEX **for** mkl_cspblas_zbsrsymv.

Array, size at least $(m * lb)$.

On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_sbsrsymv(uplo, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  REAL a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dbsrsymv(uplo, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  DOUBLE PRECISION a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_cbsrsymv(uplo, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  COMPLEX a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zbsrsymv(uplo, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_sbsrsymv(char *uplo, int *m, int *lb,
float *a, int *ia, int *ja, float *x, float *y);

void mkl_cspblas_dbsrsymv(char *uplo, int *m, int *lb,
double *a, int *ia, int *ja, double *x, double *y);

void mkl_cspblas_cbsrsymv(char *uplo, int *m, int *lb,
MKL_Complex8 *a, int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zbsrsymv(char *uplo, int *m, int *lb,
MKL_Complex16 *a, int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_cspblas_?coosymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with zero-based indexing .

Syntax

Fortran:

```
call mkl_cspblas_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_ccoosymv(uplo, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
```

C:

```
void mkl_cspblas_scoosymv (char *uplo, MKL_INT *m, float *val, MKL_INT *rowind, MKL_INT
*colind, MKL_INT *nnz, float *x, float *y);

void mkl_cspblas_dcoosymv (char *uplo, MKL_INT *m, double *val, MKL_INT *rowind,
MKL_INT *colind, MKL_INT *nnz, double *x, double *y);

void mkl_cspblas_ccoosymv (char *uplo, MKL_INT *m, MKL_Complex8 *val, MKL_INT *rowind,
MKL_INT *colind, MKL_INT *nnz, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zcoosymv (char *uplo, MKL_INT *m, MKL_Complex16 *val, MKL_INT *rowind,
MKL_INT *colind, MKL_INT *nnz, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_cspblas_?coosymv` routine performs a matrix-vector operation defined as

```
y := A*x
```

where:

x and *y* are vectors,

A is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format with zero-based indexing.

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used.
-------------	--

If $uplo = 'L'$ or ' l ', then the low triangle of the matrix A is used.

m INTEGER. Number of rows of the matrix A .

val REAL for mkl_cspblas_scoosymv.
 DOUBLE PRECISION for mkl_cspblas_dcoosymv.
 COMPLEX for mkl_cspblas_ccoosymv.
 DOUBLE COMPLEX for mkl_cspblas_zcoosymv.

Array of length nnz , contains non-zero elements of the matrix A in the arbitrary order.

Refer to $values$ array description in [Coordinate Format](#) for more details.

$rowind$ INTEGER. Array of length nnz , contains the row indices for each non-zero element of the matrix A .

Refer to $rows$ array description in [Coordinate Format](#) for more details.

$colind$ INTEGER. Array of length nnz , contains the column indices for each non-zero element of the matrix A . Refer to $columns$ array description in [Coordinate Format](#) for more details.

nnz INTEGER. Specifies the number of non-zero element of the matrix A .

Refer to nnz description in [Coordinate Format](#) for more details.

x REAL for mkl_cspblas_scoosymv.
 DOUBLE PRECISION for mkl_cspblas_dcoosymv.
 COMPLEX for mkl_cspblas_ccoosymv.
 DOUBLE COMPLEX for mkl_cspblas_zcoosymv.

Array, size is m .

On entry, the array x must contain the vector x .

Output Parameters

y REAL for mkl_cspblas_scoosymv.
 DOUBLE PRECISION for mkl_cspblas_dcoosymv.
 COMPLEX for mkl_cspblas_ccoosymv.
 DOUBLE COMPLEX for mkl_cspblas_zcoosymv.

Array, size at least m .

On exit, the array y must contain the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1    uplo
  INTEGER        m, nnz
  INTEGER        rowind(*), colind(*)
  REAL           val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1    uplo
  INTEGER        m, nnz
  INTEGER        rowind(*), colind(*)
  DOUBLE PRECISION      val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_ccoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1    uplo
  INTEGER        m, nnz
  INTEGER        rowind(*), colind(*)
  COMPLEX        val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zcoosymv(uplo, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1    uplo
  INTEGER        m, nnz
  INTEGER        rowind(*), colind(*)
  DOUBLE COMPLEX      val(*), x(*), y(*)
```

C:

```
void mkl_cspblas_scoosymv(char *uplo, int *m, float *val, int *rowind,
                           int *colind, int *nnz, float *x, float *y);

void mkl_cspblas_dcoosymv(char *uplo, int *m, double *val, int *rowind,
                           int *colind, int *nnz, double *x, double *y);

void mkl_cspblas_ccoosymv(char *uplo, int *m, MKL_Complex8 *val, int *rowind,
                           int *colind, int *nnz, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zcoosymv(char *uplo, int *m, MKL_Complex16 *val, int *rowind,
                           int *colind, int *nnz, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_cspblas_?csrtrs

Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing.

Syntax

Fortran:

```
call mkl_cspblas_scsrtrs(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_cspblas_dcsrtrs(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_cspblas_ccsrtrs(uplo, transa, diag, m, a, ia, ja, x, y)
call mkl_cspblas_zcsrtrs(uplo, transa, diag, m, a, ia, ja, x, y)
```

C:

```
void mkl_cspblas_scsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, float *a,
                           MKL_INT *ia, MKL_INT *ja, float *x, float *y);
void mkl_cspblas_dcsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, double *a,
                           MKL_INT *ia, MKL_INT *ja, double *x, double *y);
void mkl_cspblas_ccsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m,
                           MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_cspblas_zcsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m,
                           MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_cspblas_?csrtrs` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3-array variation) with zero-based indexing:

$A^*y = x$

or

$A'^*y = x,$

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.
<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then <i>A</i> * <i>y</i> = <i>x</i> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>A</i> '* <i>y</i> = <i>x</i> ,
<i>diag</i>	CHARACTER*1. Specifies whether matrix <i>A</i> is unit triangular. If <i>diag</i> = 'U' or 'u', then <i>A</i> is unit triangular. If <i>diag</i> = 'N' or 'n', then <i>A</i> is not unit triangular.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>a</i>	REAL for mkl_cspblas_scsrtrsv. DOUBLE PRECISION for mkl_cspblas_dcsrtrsv. COMPLEX for mkl_cspblas_ccsrtrsv. DOUBLE COMPLEX for mkl_cspblas_zcsrtrsv. Array containing non-zero elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>ia</i>	INTEGER. Array of length <i>m</i> +1, containing indices of elements in the array <i>a</i> , such that <i>ia</i> (<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> (<i>m</i>) is equal to the number of non-zeros. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
-----------	--

<i>ja</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i> . Its length is equal to the length of the array <i>a</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
-----------	---

NOTE

Column indices must be sorted in increasing order for each row.

<i>x</i>	REAL for mkl_cspblas_scsrtrsv. DOUBLE PRECISION for mkl_cspblas_dcsrtrsv.
----------	--

COMPLEX for `mkl_cspblas_ccsrtrsv`.
 DOUBLE COMPLEX for `mkl_cspblas_zcsrtrsv`.
Array, size is m .
 On entry, the array x must contain the vector x .

Output Parameters

y
 REAL for `mkl_cspblas_scsrtrsv`.
 DOUBLE PRECISION for `mkl_cspblas_dcsrtrsv`.
 COMPLEX for `mkl_cspblas_ccsrtrsv`.
 DOUBLE COMPLEX for `mkl_cspblas_zcsrtrsv`.
Array, size at least m .
 Contains the vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m
```

```
  INTEGER ia(*), ja(*)
```

```
  REAL a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m
```

```
  INTEGER ia(*), ja(*)
```

```
  DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_ccsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m
```

```
  INTEGER ia(*), ja(*)
```

```
  COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zcsrtrsv(uplo, transa, diag, m, a, ia, ja, x, y)
```

```
  CHARACTER*1 uplo, transa, diag
```

```
  INTEGER m
```

```
  INTEGER ia(*), ja(*)
```

```
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_scsrtrs(char *uplo, char *transa, char *diag, int *m,
float *a, int *ia, int *ja, float *x, float *y);

void mkl_cspblas_dcsrtrs(char *uplo, char *transa, char *diag, int *m,
double *a, int *ia, int *ja, double *x, double *y);

void mkl_cspblas_ccsrtrs(char *uplo, char *transa, char *diag, int *m,
MKL_Complex8 *a, int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zcsrtrs(char *uplo, char *transa, char *diag, int *m,
MKL_Complex16 *a, int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_cspblas_?bsrtrs

Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing.

Syntax**Fortran:**

```
call mkl_cspblas_sbsrtrs(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_cspblas_dbsrtrs(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_cspblas_cbsrtrs(uplo, transa, diag, m, lb, a, ia, ja, x, y)
call mkl_cspblas_zbsrtrs(uplo, transa, diag, m, lb, a, ia, ja, x, y)
```

C:

```
void mkl_cspblas_sbsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_INT
*lb, float *a, MKL_INT *ia, MKL_INT *ja, float *x, float *y);
void mkl_cspblas_dbsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_INT
*lb, double *a, MKL_INT *ia, MKL_INT *ja, double *x, double *y);
void mkl_cspblas_cbsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_INT
*lb, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_cspblas_zbsrtrs (char *uplo, char *transa, char *diag, MKL_INT *m, MKL_INT
*lb, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_cspblas_?bsrtrs` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing:

`y := A*x`

or

`y := A'*x,`

where:

x and *y* are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	CHARACTER*1. Specifies the upper or low triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix <i>A</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix <i>A</i> is used.
<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as <i>y</i> := <i>A</i> * <i>x</i> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as <i>y</i> := <i>A'</i> * <i>x</i> .
<i>diag</i>	CHARACTER*1. Specifies whether matrix <i>A</i> is unit triangular or not. If <i>diag</i> = 'U' or 'u', <i>A</i> is unit triangular. If <i>diag</i> = 'N' or 'n', <i>A</i> is not unit triangular.
<i>m</i>	INTEGER. Number of block rows of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>a</i>	REAL for mkl_cspblas_sbsrtrsv. DOUBLE PRECISION for mkl_cspblas_dbsrtrsv. COMPLEX for mkl_cspblas_cbsrtrsv. DOUBLE COMPLEX for mkl_cspblas_zbsrtrsv. Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb</i> * <i>lb</i> . Refer to <i>values</i> array description in BSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>ia</i>	INTEGER. Array of length $(m + 1)$, containing indices of block in the array <i>a</i> , such that $ia(I)$ is the index in the array <i>a</i> of the first non-zero element from the row I . The value of the last element $ia(m + 1)$ is equal to the number of non-zero blocks. Refer to <i>rowIndex</i> array description in BSR Format for more details.
<i>ja</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks of the matrix <i>A</i> . Refer to <i>columns</i> array description in BSR Format for more details.
<i>x</i>	REAL for <code>mkl_cspblas_sbsrtrsv</code> . DOUBLE PRECISION for <code>mkl_cspblas_dbsrtrsv</code> . COMPLEX for <code>mkl_cspblas_cbsrtrsv</code> . DOUBLE COMPLEX for <code>mkl_cspblas_zbsrtrsv</code> . Array, size $(m * lb)$. On entry, the array <i>x</i> must contain the vector <i>x</i> .

Output Parameters

<i>y</i>	REAL for <code>mkl_cspblas_sbsrtrsv</code> . DOUBLE PRECISION for <code>mkl_cspblas_dbsrtrsv</code> . COMPLEX for <code>mkl_cspblas_cbsrtrsv</code> . DOUBLE COMPLEX for <code>mkl_cspblas_zbsrtrsv</code> . Array, size at least $(m * lb)$. On exit, the array <i>y</i> must contain the vector <i>y</i> .
----------	--

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_sbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  REAL a(*), x(*), y(*)

SUBROUTINE mkl_cspblas_dbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  DOUBLE PRECISION a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_cbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  COMPLEX a(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_zbsrtrsv(uplo, transa, diag, m, lb, a, ia, ja, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, lb
  INTEGER ia(*), ja(*)
  DOUBLE COMPLEX a(*), x(*), y(*)
```

C:

```
void mkl_cspblas_sbsrtrsv(char *uplo, char *transa, char *diag, int *m,
                           int *lb, float *a, int *ia, int *ja, float *x, float *y);

void mkl_cspblas_dbsrtrsv(char *uplo, char *transa, char *diag, int *m,
                           int *lb, double *a, int *ia, int *ja, double *x, double *y);

void mkl_cspblas_cbsrtrsv(char *uplo, char *transa, char *diag, int *m,
                           int *lb, MKL_Complex8 *a, int *ia, int *ja, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zbsrtrsv(char *uplo, char *transa, char *diag, int *m,
                           int *lb, MKL_Complex16 *a, int *ia, int *ja, MKL_Complex16 *x, MKL_Complex16 *y);
```

[mkl_cspblas_?cootrv](#)

Triangular solvers with simplified interface for a sparse matrix in the coordinate format with zero-based indexing .

Syntax

Fortran:

```
call mkl_cspblas_scootrv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_dcootrv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_ccootrv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
call mkl_cspblas_zcootrv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
```

C:

```
void mkl_cspblas_scootrv (char *uplo, char *transa, char *diag, MKL_INT *m, float
                           *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, float *x, float *y);
void mkl_cspblas_dcootrv (char *uplo, char *transa, char *diag, MKL_INT *m, double
                           *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, double *x, double *y);
void mkl_cspblas_ccootrv (char *uplo, char *transa, char *diag, MKL_INT *m,
                           MKL_Complex8 *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, MKL_Complex8 *x,
                           MKL_Complex8 *y);
```

```
void mkl_cspblas_zcootrs (char *uplo, char *transa, char *diag, MKL_INT *m,
MKL_Complex16 *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, MKL_Complex16 *x,
MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_cspblas_?cootrs` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format with zero-based indexing:

$A^*y = x$

or

$A'^*y = x$,

where:

x and y are vectors,

A is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, A' is the transpose of A .

NOTE

This routine supports only zero-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or low triangle of the matrix A is considered. If <i>uplo</i> = 'U' or 'u', then the upper triangle of the matrix A is used. If <i>uplo</i> = 'L' or 'l', then the low triangle of the matrix A is used.
<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $A^*y = x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $A'^*y = x$,
<i>diag</i>	CHARACTER*1. Specifies whether A is unit triangular. If <i>diag</i> = 'U' or 'u', then A is unit triangular. If <i>diag</i> = 'N' or 'n', then A is not unit triangular.
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>val</i>	REAL for <code>mkl_cspblas_scootrs</code> . DOUBLE PRECISION for <code>mkl_cspblas_dcootrs</code> . COMPLEX for <code>mkl_cspblas_ccootrs</code> . DOUBLE COMPLEX for <code>mkl_cspblas_zcootrs</code> .

Array of length nnz , contains non-zero elements of the matrix A in the arbitrary order.

Refer to *values* array description in [Coordinate Format](#) for more details.

rowind

INTEGER. Array of length nnz , contains the row indices for each non-zero element of the matrix A .

Refer to *rows* array description in [Coordinate Format](#) for more details.

colind

INTEGER. Array of length nnz , contains the column indices for each non-zero element of the matrix A . Refer to *columns* array description in [Coordinate Format](#) for more details.

nnz

INTEGER. Specifies the number of non-zero element of the matrix A .

Refer to *nnz* description in [Coordinate Format](#) for more details.

x

REAL **for** mkl_cspblas_scootrs.

DOUBLE PRECISION **for** mkl_cspblas_dcootrs.

COMPLEX **for** mkl_cspblas_ccootrs.

DOUBLE COMPLEX **for** mkl_cspblas_zcootrs.

Array, size is m .

On entry, the array *x* must contain the vector *x*.

Output Parameters

y

REAL **for** mkl_cspblas_scootrs.

DOUBLE PRECISION **for** mkl_cspblas_dcootrs.

COMPLEX **for** mkl_cspblas_ccootrs.

DOUBLE COMPLEX **for** mkl_cspblas_zcootrs.

Array, size at least m .

Contains the vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_cspblas_scootrs(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
CHARACTER*1 uplo, transa, diag
INTEGER m, nnz
INTEGER rowind(*), colind(*)
REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cspblas_dcootrv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, nnz
  INTEGER rowind(*), colind(*)
  DOUBLE PRECISION val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_ccootrv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, nnz
  INTEGER rowind(*), colind(*)
  COMPLEX val(*), x(*), y(*)

SUBROUTINE mkl_cspblas_zcootrv(uplo, transa, diag, m, val, rowind, colind, nnz, x, y)
  CHARACTER*1 uplo, transa, diag
  INTEGER m, nnz
  INTEGER rowind(*), colind(*)
  DOUBLE COMPLEX val(*), x(*), y(*)
```

C:

```
void mkl_cspblas_scootrv(char *uplo, char *transa, char *diag, int *m,
float *val, int *rowind, int *colind, int *nnz, float *x, float *y);

void mkl_cspblas_dcootrv(char *uplo, char *transa, char *diag, int *m,
double *val, int *rowind, int *colind, int *nnz, double *x, double *y);

void mkl_cspblas_ccootrv(char *uplo, char *transa, char *diag, int *m,
MKL_Complex8 *val, int *rowind, int *colind, int *nnz, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zcootrv(char *uplo, char *transa, char *diag, int *m,
MKL_Complex16 *val, int *rowind, int *colind, int *nnz, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?csrmv

Computes matrix - vector product of a sparse matrix stored in the CSR format.

Syntax**Fortran:**

```
call mkl_scsrsv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_dcsrsv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_ccsrsv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_zcsrsv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
```

C:

```

void mkl_scsrmv (char *transa, MKL_INT *m, MKL_INT *k, float *alpha, char *matdescra,
float *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, float *x, float *beta,
float *y);

void mkl_dcsrmv (char *transa, MKL_INT *m, MKL_INT *k, double *alpha, char *matdescra,
double *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, double *x, double *beta,
double *y);

void mkl_ccsrmv (char *transa, MKL_INT *m, MKL_INT *k, MKL_Complex8 *alpha, char
*matdescra, MKL_Complex8 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex8 *x, MKL_Complex8 *beta, MKL_Complex8 *y);

void mkl_zcsrmv (char *transa, MKL_INT *m, MKL_INT *k, MKL_Complex16 *alpha, char
*matdescra, MKL_Complex16 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex16 *x, MKL_Complex16 *beta, MKL_Complex16 *y);

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?csrmv` routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A'*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in the CSR format, *A'* is the transpose of *A*.

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation.
	If <i>transa</i> = 'N' or 'n', then <i>y</i> := <i>alpha</i> * <i>A</i> * <i>x</i> + <i>beta</i> * <i>y</i>
	If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>y</i> := <i>alpha</i> * <i>A'</i> * <i>x</i> + <i>beta</i> * <i>y</i> ,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_scsrmv</code> .
	DOUBLE PRECISION for <code>mkl_dcsrmv</code> .

COMPLEX for `mkl_ccsrmv`.
 DOUBLE COMPLEX for `mkl_zcsrmv`.
 Specifies the scalar *alpha*.

matdescra
 CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in [Table "Possible Values of the Parameter *matdescra* \(*descra*\)"](#). Possible combinations of element values of this parameter are given in [Table "Possible Combinations of Element Values of the Parameter *matdescra*"](#).

val
 REAL for `mkl_scsrmv`.
 DOUBLE PRECISION for `mkl_dcsrmv`.
 COMPLEX for `mkl_ccsrmv`.
 DOUBLE COMPLEX for `mkl_zcsrmv`.
 Array containing non-zero elements of the matrix *A*.
 For one-based indexing its length is $pntre(m) - pntrb(1)$.
 For zero-based indexing its length is $pntre(m-1) - pntrb(0)$.
 Refer to *values* array description in [CSR Format](#) for more details.

indx
 INTEGER. Array containing the column indices for each non-zero element of the matrix *A*. Its length is equal to length of the *val* array.
 Refer to *columns* array description in [CSR Format](#) for more details.

pntrb
 INTEGER. Array of length *m*.
 For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of row *i* in the arrays *val* and *indx*.
 For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of row *i* in the arrays *val* and *indx*. Refer to *pointerb* array description in [CSR Format](#) for more details.

pntre
 INTEGER. Array of length *m*.
 For one-based indexing this array contains row indices, such that $pntre(i) - pntrb(1)$ is the last index of row *i* in the arrays *val* and *indx*.
 For zero-based indexing this array contains row indices, such that $pntre(i) - pntrb(0)-1$ is the last index of row *i* in the arrays *val* and *indx*. Refer to *pointerE* array description in [CSR Format](#) for more details.

x
 REAL for `mkl_scsrmv`.
 DOUBLE PRECISION for `mkl_dcsrmv`.
 COMPLEX for `mkl_ccsrmv`.
 DOUBLE COMPLEX for `mkl_zcsrmv`.
 Array, size at least *k* if *transa* = 'N' or 'n' and at least *m* otherwise. On entry, the array *x* must contain the vector *x*.

beta
 REAL for `mkl_scsrmv`.

DOUBLE PRECISION for mkl_dcsrmv.
 COMPLEX for mkl_ccsrmv.
 DOUBLE COMPLEX for mkl_zcsrmv.
Specifies the scalar *beta*.

y REAL for mkl_scsrmv.
 DOUBLE PRECISION for mkl_dcsrmv.
 COMPLEX for mkl_ccsrmv.
 DOUBLE COMPLEX for mkl_zcsrmv.
 Array, size at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y Overwritten by the updated vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrmv(transa, m, k, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
CHARACTER*1 transa
CHARACTER matdescra(*)
INTEGER m, k
INTEGER indx(*), pntrb(m), pntre(m)
REAL alpha, beta
REAL val(*), x(*), y(*)

SUBROUTINE mkl_dcsrmv(transa, m, k, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
CHARACTER*1 transa
CHARACTER matdescra(*)
INTEGER m, k
INTEGER indx(*), pntrb(m), pntre(m)
DOUBLE PRECISION alpha, beta
DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_ccsrmv(transa, m, k, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
COMPLEX alpha, beta
```

```
COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zcsrmv(transa, m, k, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, k
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE COMPLEX alpha, beta
```

```
DOUBLE COMPLEX val(*), x(*), y(*)
```

C:

```
void mkl_scsrsv(char *transa, int *m, int *k, float *alpha, char *matdescra,
float *val, int *indx, int *pntrb, int *pntre, float *x, float *beta, float *y);
void mkl_dcsrsv(char *transa, int *m, int *k, double *alpha, char *matdescra,
double *val, int *indx, int *pntrb, int *pntre, double *x, double *beta, double *y);
void mkl_ccsrsv(char *transa, int *m, int *k, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, int *indx, int *pntrb, int *pntre, MKL_Complex8 *x, MKL_Complex8 *beta, double *y);
void mkl_zcsrsv(char *transa, int *m, int *k, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, int *indx, int *pntrb, int *pntre, MKL_Complex16 *x, MKL_Complex16 *beta,
MKL_Complex16 *y);
```

mkl_?bsrmv

Computes matrix - vector product of a sparse matrix stored in the BSR format.

Syntax

Fortran:

```
call mkl_sbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta,
y)
```

```
call mkl_dbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta,
y)
```

```
call mkl_cbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta,
y)
```

```
call mkl_zbsrmv(transa, m, k, lb, alpha, matdescra, val, indx, pntrb, pntre, x, beta,
y)
```

C:

```
void mkl_sbsrmv (char *transa, MKL_INT *m, MKL_INT *k, MKL_INT *lb, float *alpha, char
*matdescra, float *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, float *x, float
*beta, float *y);
```

```
void mkl_dbsrmv (char *transa, MKL_INT *m, MKL_INT *k, MKL_INT *lb, double *alpha,
char *matdescra, double *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, double
*x, double *beta, double *y);
```

```
void mkl_cbsrmv (char *transa, MKL_INT *m, MKL_INT *k, MKL_INT *lb, MKL_Complex8
*alpha, char *matdescra, MKL_Complex8 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT
*pntre, MKL_Complex8 *x, MKL_Complex8 *beta, MKL_Complex8 *y);
```

```
void mkl_zbsrmv (char *transa, MKL_INT *m, MKL_INT *k, MKL_INT *lb, MKL_Complex16
*alpha, char *matdescra, MKL_Complex16 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT
*pntre, MKL_Complex16 *x, MKL_Complex16 *beta, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?bsrmv` routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A'*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* block sparse matrix in the BSR format, *A'* is the transpose of *A*.

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-vector product is computed as <i>y</i> := <i>alpha</i> * <i>A</i> * <i>x</i> + <i>beta</i> * <i>y</i>
---------------	--

If $\text{transa} = \text{'T'}$ or 't' or 'C' or 'c' , then the matrix-vector product is computed as $y := \alpha A^*x + \beta y$,

<i>m</i>	INTEGER. Number of block rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of block columns of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_sbsrmv. DOUBLE PRECISION for mkl_dbsrmv. COMPLEX for mkl_cbsrmv. DOUBLE COMPLEX for mkl_zbsrmv. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for mkl_sbsrmv. DOUBLE PRECISION for mkl_dbsrmv. COMPLEX for mkl_cbsrmv. DOUBLE COMPLEX for mkl_zbsrmv. Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by $lb * lb$. Refer to <i>values</i> array description in BSR Format for more details.
<i>indx</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> . Refer to <i>columns</i> array description in BSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i> . For one-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of block row <i>i</i> in the array <i>indx</i> . For zero-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of block row <i>i</i> in the array <i>indx</i> . Refer to <i>pointerB</i> array description in BSR Format for more details.
<i>pntre</i>	INTEGER. Array of length <i>m</i> . For one-based indexing this array contains row indices, such that $pntre(i) - pntrb(1)$ is the last index of block row <i>i</i> in the array <i>indx</i> . For zero-based indexing this array contains row indices, such that $pntre(i) - pntrb(0) - 1$ is the last index of block row <i>i</i> in the array <i>indx</i> .

Refer to *pointerE* array description in [BSR Format](#) for more details.

x

REAL for mkl_sbsrmv.
 DOUBLE PRECISION for mkl_dbsrmv.
 COMPLEX for mkl_cbsrmv.
 DOUBLE COMPLEX for mkl_zbsrmv.
 Array, size at least ($k * lb$) if *transa* = 'N' or 'n', and at least ($m * lb$) otherwise. On entry, the array *x* must contain the vector *x*.

beta

REAL for mkl_sbsrmv.
 DOUBLE PRECISION for mkl_dbsrmv.
 COMPLEX for mkl_cbsrmv.
 DOUBLE COMPLEX for mkl_zbsrmv.
 Specifies the scalar *beta*.

y

REAL for mkl_sbsrmv.
 DOUBLE PRECISION for mkl_dbsrmv.
 COMPLEX for mkl_cbsrmv.
 DOUBLE COMPLEX for mkl_zbsrmv.
 Array, size at least ($m * lb$) if *transa* = 'N' or 'n', and at least ($k * lb$) otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y Overwritten by the updated vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
CHARACTER*1 transa
CHARACTER matdescra(*)
INTEGER m, k, lb
INTEGER indx(*), pntrb(m), pntre(m)
REAL alpha, beta
REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
```

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, k, lb

INTEGER indx(*), pntrb(m), pntre(m)

DOUBLE PRECISION alpha, beta

DOUBLE PRECISION val(*), x(*), y(*)

```
SUBROUTINE mkl_cbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
```

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, k, lb

INTEGER indx(*), pntrb(m), pntre(m)

COMPLEX alpha, beta

COMPLEX val(*), x(*), y(*)

```
SUBROUTINE mkl_zbsrmv(transa, m, k, lb, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
```

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, k, lb

INTEGER indx(*), pntrb(m), pntre(m)

DOUBLE COMPLEX alpha, beta

DOUBLE COMPLEX val(*), x(*), y(*)

C:

```
void mkl_sbsrmv(char *transa, int *m, int *k, int *lb,
float *alpha, char *matdescra, float *val, int *indx,
int *pntrb, int *pntre, float *x, float *beta, float *y);
```

```
void mkl_dbsrmv(char *transa, int *m, int *k, int *lb,
double *alpha, char *matdescra, double *val, int *indx,
int *pntrb, int *pntre, double *x, double *beta, double *y);
```

```
void mkl_cbsrmv(char *transa, int *m, int *k, int *lb,
MKL_Complex8 *alpha, char *matdescra, MKL_Complex8 *val, int *indx,
int *pntrb, int *pntre, MKL_Complex8 *x, MKL_Complex8 *beta, MKL_Complex8 *y);
```

```
void mkl_zbsrmv(char *transa, int *m, int *k, int *lb,
                  MKL_Complex16 *alpha, char *matdescra, MKL_Complex16 *val, int *indx,
                  int *pntrb, int *pntre, MKL_Complex16 *x, MKL_Complex16 *beta, MKL_Complex16 *y);
```

mkl_?cscmv

Computes matrix-vector product for a sparse matrix in the CSC format.

Syntax

Fortran:

```
call mkl_scscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_dcscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_ccscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
call mkl_zcscmv(transa, m, k, alpha, matdescra, val, indx, pntrb, pntre, x, beta, y)
```

C:

```
void mkl_scscmv (char *transa, MKL_INT *m, MKL_INT *k, float *alpha, char *matdescra,
float *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, float *x, float *beta,
float *y);

void mkl_dcscmv (char *transa, MKL_INT *m, MKL_INT *k, double *alpha, char *matdescra,
double *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, double *x, double *beta,
double *y);

void mkl_ccscmv (char *transa, MKL_INT *m, MKL_INT *k, MKL_Complex8 *alpha, char
*matdescra, MKL_Complex8 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex8 *x, MKL_Complex8 *beta, MKL_Complex8 *y);

void mkl_zcscmv (char *transa, MKL_INT *m, MKL_INT *k, MKL_Complex16 *alpha, char
*matdescra, MKL_Complex16 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex16 *x, MKL_Complex16 *beta, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?cscmv` routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A'*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in compressed sparse column (CSC) format, *A'* is the transpose of *A*.

NOTE

This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha A^T x + \beta y$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_scscmv. DOUBLE PRECISION for mkl_dcscmv. COMPLEX for mkl_ccscmv. DOUBLE COMPLEX for mkl_zcscmv. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for mkl_scscmv. DOUBLE PRECISION for mkl_dcscmv. COMPLEX for mkl_ccscmv. DOUBLE COMPLEX for mkl_zcscmv. Array containing non-zero elements of the matrix <i>A</i> . For one-based indexing its length is $\text{pntrb}(k) - \text{pntrb}(1)$. For zero-based indexing its length is $\text{pntrb}(m-1) - \text{pntrb}(0)$. Refer to <i>values</i> array description in CSC Format for more details.
<i>indx</i>	INTEGER. Array containing the row indices for each non-zero element of the matrix <i>A</i> . Its length is equal to length of the <i>val</i> array. Refer to <i>rows</i> array description in CSC Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>k</i> . For one-based indexing this array contains column indices, such that $\text{pntrb}(i) - \text{pntrb}(1) + 1$ is the first index of column <i>i</i> in the arrays <i>val</i> and <i>indx</i> .

For zero-based indexing this array contains column indices, such that $pntrb(i) - pntrb(0)$ is the first index of column i in the arrays `val` and `indx`.

Refer to `pointerb` array description in [CSC Format](#) for more details.

`pntre`

INTEGER. Array of length k .

For one-based indexing this array contains column indices, such that $pntre(i) - pntre(1)$ is the last index of column i in the arrays `val` and `indx`.

For zero-based indexing this array contains column indices, such that $pntre(i) - pntre(1) - 1$ is the last index of column i in the arrays `val` and `indx`.

Refer to `pointerE` array description in [CSC Format](#) for more details.

`x`

REAL **for** `mkl_scscmv`.

DOUBLE PRECISION **for** `mkl_dcscmv`.

COMPLEX **for** `mkl_ccscmv`.

DOUBLE COMPLEX **for** `mkl_zcscmv`.

Array, size at least k if $transa = 'N'$ or ' n ' and at least m otherwise. On entry, the array `x` must contain the vector x .

`beta`

REAL **for** `mkl_scscmv`.

DOUBLE PRECISION **for** `mkl_dcscmv`.

COMPLEX **for** `mkl_ccscmv`.

DOUBLE COMPLEX **for** `mkl_zcscmv`.

Specifies the scalar `beta`.

`y`

REAL **for** `mkl_scscmv`.

DOUBLE PRECISION **for** `mkl_dcscmv`.

COMPLEX **for** `mkl_ccscmv`.

DOUBLE COMPLEX **for** `mkl_zcscmv`.

Array, size at least m if $transa = 'N'$ or ' n ' and at least k otherwise. On entry, the array `y` must contain the vector y .

Output Parameters

`y`

Overwritten by the updated vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scscmv(transa, m, k, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
```

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, k, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

REAL alpha, beta

REAL val(*), x(*), y(*)

```
SUBROUTINE mkl_dcscmv(transa, m, k, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
```

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, k, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

DOUBLE PRECISION alpha, beta

DOUBLE PRECISION val(*), x(*), y(*)

```
SUBROUTINE mkl_ccscmv(transa, m, k, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
```

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, k, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

COMPLEX alpha, beta

COMPLEX val(*), x(*), y(*)

```
SUBROUTINE mkl_zcscmv(transa, m, k, alpha, matdescra, val, indx,
pntrb, pntre, x, beta, y)
```

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, k, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

DOUBLE COMPLEX alpha, beta

DOUBLE COMPLEX val(*), x(*), y(*)

C:

```
void mkl_scscmv(char *transa, int *m, int *k, float *alpha,
char *matdescra, float *val, int *indx, int *pntrb,
int *pntre, float *x, float *beta, float *y);

void mkl_dcscmv(char *transa, int *m, int *k, double *alpha,
char *matdescra, double *val, int *indx, int *pntrb,
int *pntre, double *x, double *beta, double *y);

void mkl_ccscmv(char *transa, int *m, int *k, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, int *indx, int *pntrb,
int *pntre, MKL_Complex8 *x, MKL_Complex8 *beta, MKL_Complex8 *y);

void mkl_zcscmv(char *transa, int *m, int *k, MKL_Complex16 *alpha,
char *matdescra, MKL_Complex16 *val, int *indx, int *pntrb,
int *pntre, MKL_Complex16 *x, MKL_Complex16 *beta, MKL_Complex16 *y);
```

mkl_?coomv

*Computes matrix - vector product for a sparse matrix
in the coordinate format.*

Syntax**Fortran:**

```
call mkl_scoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
call mkl_dcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
call mkl_ccoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
call mkl_zcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
```

C:

```
void mkl_scoomv (char *transa, MKL_INT *m, MKL_INT *k, float *alpha, char *matdescra,
float *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, float *x, float *beta,
float *y);

void mkl_dcoomv (char *transa, MKL_INT *m, MKL_INT *k, double *alpha, char *matdescra,
double *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, double *x, double *beta,
double *y);

void mkl_ccoomv (char *transa, MKL_INT *m, MKL_INT *k, MKL_Complex8 *alpha, char
*matdescra, MKL_Complex8 *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz,
MKL_Complex8 *x, MKL_Complex8 *beta, MKL_Complex8 *y);

void mkl_zcoomv (char *transa, MKL_INT *m, MKL_INT *k, MKL_Complex16 *alpha, char
*matdescra, MKL_Complex16 *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz,
MKL_Complex16 *x, MKL_Complex16 *beta, MKL_Complex16 *y);
```

Include Files

- **Fortran:** mkl.fi
- **C:** mkl.h

Description

The `mkl_?coomv` routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A'*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix in compressed coordinate format, *A'* is the transpose of *A*.

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $y := \alpha A x + \beta y$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha A' x + \beta y$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_scoomv</code> . DOUBLE PRECISION for <code>mkl_dcoomv</code> . COMPLEX for <code>mkl_ccoomv</code> . DOUBLE COMPLEX for <code>mkl_zcoomv</code> . Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for <code>mkl_scoomv</code> . DOUBLE PRECISION for <code>mkl_dcoomv</code> . COMPLEX for <code>mkl_ccoomv</code> . DOUBLE COMPLEX for <code>mkl_zcoomv</code> . Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order.

Refer to `values` array description in [Coordinate Format](#) for more details.

`rowind`

INTEGER. Array of length nnz , contains the row indices for each non-zero element of the matrix A .

Refer to `rows` array description in [Coordinate Format](#) for more details.

`colind`

INTEGER. Array of length nnz , contains the column indices for each non-zero element of the matrix A . Refer to `columns` array description in [Coordinate Format](#) for more details.

`nnz`

INTEGER. Specifies the number of non-zero element of the matrix A .

Refer to `nnz` description in [Coordinate Format](#) for more details.

`x`

REAL for `mkl_scoomv`.

DOUBLE PRECISION for `mkl_dcoomv`.

COMPLEX for `mkl_ccoomv`.

DOUBLE COMPLEX for `mkl_zcoomv`.

Array, size at least k if $transa = 'N'$ or ' n ' and at least m otherwise. On entry, the array x must contain the vector x .

`beta`

REAL for `mkl_scoomv`.

DOUBLE PRECISION for `mkl_dcoomv`.

COMPLEX for `mkl_ccoomv`.

DOUBLE COMPLEX for `mkl_zcoomv`.

Specifies the scalar β .

`y`

REAL for `mkl_scoomv`.

DOUBLE PRECISION for `mkl_dcoomv`.

COMPLEX for `mkl_ccoomv`.

DOUBLE COMPLEX for `mkl_zcoomv`.

Array, size at least m if $transa = 'N'$ or ' n ' and at least k otherwise. On entry, the array y must contain the vector y .

Output Parameters

`y`

Overwritten by the updated vector y .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, k, nnz
  INTEGER rowind(*), colind(*)
  REAL alpha, beta
  REAL val(*), x(*), y(*)

SUBROUTINE mkl_dcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, k, nnz
  INTEGER rowind(*), colind(*)
  DOUBLE PRECISION alpha, beta
  DOUBLE PRECISION val(*), x(*), y(*)

SUBROUTINE mkl_ccoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, k, nnz
  INTEGER rowind(*), colind(*)
  COMPLEX alpha, beta
  COMPLEX val(*), x(*), y(*)

SUBROUTINE mkl_zcoomv(transa, m, k, alpha, matdescra, val, rowind, colind, nnz, x, beta, y)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, k, nnz
  INTEGER rowind(*), colind(*)
  DOUBLE COMPLEX alpha, beta
  DOUBLE COMPLEX val(*), x(*), y(*)
```

C:

```
void mkl_scoomv(char *transa, int *m, int *k, float *alpha, char *matdescra,
float *val, int *rowind, int *colind, int *nnz, float *x, float *beta, float *y);
void mkl_dcoomv(char *transa, int *m, int *k, double *alpha, char *matdescra,
double *val, int *rowind, int *colind, int *nnz, double *x, double *beta, double *y);
```

```
void mkl_ccoomv(char *transa, int *m, int *k, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, int *rowind, int *colind, int *nnz, MKL_Complex8 *x, MKL_Complex8 *beta,
MKL_Complex8 *y);

void mkl_zcoomv(char *transa, int *m, int *k, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, int *rowind, int *colind, int *nnz, MKL_Complex16 *x, MKL_Complex16 *beta,
MKL_Complex16 *y);
```

mkl_?csrv

Solves a system of linear equations for a sparse matrix in the CSR format.

Syntax

Fortran:

```
call mkl_scsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_dcsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_ccsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_zcsrsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

C:

```
void mkl_scsrsv (char *transa, MKL_INT *m, float *alpha, char *matdescra, float *val,
MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, float *x, float *y);
void mkl_dcsrsv (char *transa, MKL_INT *m, double *alpha, char *matdescra, double *val,
MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, double *x, double *y);
void mkl_ccsrsv (char *transa, MKL_INT *m, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, MKL_Complex8 *x,
MKL_Complex8 *y);
void mkl_zcsrsv (char *transa, MKL_INT *m, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, MKL_Complex16 *x,
MKL_Complex16 *y);
```

Include Files

- **Fortran:** mkl.fi
- **C:** mkl.h

Description

The `mkl_?csrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSR format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A')*x,
```

where:

`alpha` is scalar, `x` and `y` are vectors, `A` is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, `A'` is the transpose of `A`.

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then $y := \alpha^{-1}(A)x$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $y := \alpha^{-1}(A^T)x$,
<i>m</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_scsrsv. DOUBLE PRECISION for mkl_dcsrsv. COMPLEX for mkl_ccsrsv. DOUBLE COMPLEX for mkl_zcsrsv. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for mkl_scsrsv. DOUBLE PRECISION for mkl_dcsrsv. COMPLEX for mkl_ccsrsv. DOUBLE COMPLEX for mkl_zcsrsv. Array containing non-zero elements of the matrix <i>A</i> . For one-based indexing its length is $pntre(m) - pntrb(1)$. For zero-based indexing its length is $pntre(m-1) - pntrb(0)$. Refer to <i>values</i> array description in CSR Format for more details.
<i>indx</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i> . Its length is equal to length of the <i>val</i> array. Refer to <i>columns</i> array description in CSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

NOTE

Column indices must be sorted in increasing order for each row.

pntrb

INTEGER. Array of length m .

For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of row i in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of row i in the arrays *val* and *indx*. Refer to *pointerb* array description in [CSR Format](#) for more details.

pntre

INTEGER. Array of length m .

For one-based indexing this array contains row indices, such that $pntre(i) - pntrb(1)$ is the last index of row i in the arrays *val* and *indx*.

For zero-based indexing this array contains row indices, such that $pntre(i) - pntrb(0) - 1$ is the last index of row i in the arrays *val* and *indx*. Refer to *pointerE* array description in [CSR Format](#) for more details.

x

REAL **for** mkl_scsrsv.

DOUBLE PRECISION **for** mkl_dcsrsv.

COMPLEX **for** mkl_ccsrsv.

DOUBLE COMPLEX **for** mkl_zcsrsv.

Array, size at least m .

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y

REAL **for** mkl_scsrsv.

DOUBLE PRECISION **for** mkl_dcsrsv.

COMPLEX **for** mkl_ccsrsv.

DOUBLE COMPLEX **for** mkl_zcsrsv.

Array, size at least m .

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y

Contains solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  REAL alpha
```

```
  REAL val(*)
```

```
  REAL x(*), y(*)
```

```
SUBROUTINE mkl_dcsrv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  DOUBLE PRECISION alpha
```

```
  DOUBLE PRECISION val(*)
```

```
  DOUBLE PRECISION x(*), y(*)
```

```
SUBROUTINE mkl_ccsrv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  COMPLEX alpha
```

```
  COMPLEX val(*)
```

```
  COMPLEX x(*), y(*)
```

```
SUBROUTINE mkl_zcsrv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  DOUBLE COMPLEX alpha
```

```
  DOUBLE COMPLEX val(*)
```

```
  DOUBLE COMPLEX x(*), y(*)
```

C:

```
void mkl_scsrsv(char *transa, int *m, float *alpha, char *matdescra,
float *val, int *indx, int *pntrb, int *pntre, float *x, float *y);

void mkl_dcsrsv(char *transa, int *m, double *alpha, char *matdescra,
double *val, int *indx, int *pntrb, int *pntre, double *x, double *y);

void mkl_ccsrsv(char *transa, int *m, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, int *indx, int *pntrb, int *pntre, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zcsrsv(char *transa, int *m, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, int *indx, int *pntrb, int *pntre, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?bsrsv

Solves a system of linear equations for a sparse matrix in the BSR format.

Syntax**Fortran:**

```
call mkl_sbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_dbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_cbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_zbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

C:

```
void mkl_sbsrsv (char *transa, MKL_INT *m, MKL_INT *lb, float *alpha, char *matdescra,
float *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, float *x, float *y);

void mkl_dbsrsv (char *transa, MKL_INT *m, MKL_INT *lb, double *alpha, char *matdescra,
double *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, double *x, double *y);

void mkl_cbsrsv (char *transa, MKL_INT *m, MKL_INT *lb, MKL_Complex8 *alpha, char
*matdescra, MKL_Complex8 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zbsrsv (char *transa, MKL_INT *m, MKL_INT *lb, MKL_Complex16 *alpha, char
*matdescra, MKL_Complex16 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The mkl_?bsrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the BSR format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A')* x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then <i>y</i> := <i>alpha</i> *inv(<i>A</i>)* <i>x</i> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>y</i> := <i>alpha</i> *inv(<i>A'</i>)* <i>x</i> ,
<i>m</i>	INTEGER. Number of block columns of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_sbsrv. DOUBLE PRECISION for mkl_dbsrv. COMPLEX for mkl_cbsrv. DOUBLE COMPLEX for mkl_zbsrv. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for mkl_sbsrv. DOUBLE PRECISION for mkl_dbsrv. COMPLEX for mkl_cbsrv. DOUBLE COMPLEX for mkl_zbsrv. Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb</i> * <i>lb</i> . Refer to the <i>values</i> array description in BSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

<i>indx</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix A . Its length is equal to the number of non-zero blocks in the matrix A .
	Refer to the <i>columns</i> array description in BSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length m .
	For one-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of block row i in the array <i>indx</i> .
	For zero-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of block row i in the array <i>indx</i>
	Refer to <i>pointerB</i> array description in BSR Format for more details.
<i>pntre</i>	INTEGER. Array of length m .
	For one-based indexing this array contains row indices, such that $pntre(i) - pntrb(1)$ is the last index of block row i in the array <i>indx</i> .
	For zero-based indexing this array contains row indices, such that $pntre(i) - pntrb(0) - 1$ is the last index of block row i in the array <i>indx</i> .
	Refer to <i>pointerE</i> array description in BSR Format for more details.
<i>x</i>	REAL for <code>mkl_sbsrsv</code> . DOUBLE PRECISION for <code>mkl_dbsrsv</code> . COMPLEX for <code>mkl_cbsrsv</code> . DOUBLE COMPLEX for <code>mkl_zbsrsv</code> . Array, size at least $(m * lb)$. On entry, the array <i>x</i> must contain the vector x . The elements are accessed with unit increment.
<i>y</i>	REAL for <code>mkl_sbsrsv</code> . DOUBLE PRECISION for <code>mkl_dbsrsv</code> . COMPLEX for <code>mkl_cbsrsv</code> . DOUBLE COMPLEX for <code>mkl_zbsrsv</code> . Array, size at least $(m * lb)$. On entry, the array <i>y</i> must contain the vector y . The elements are accessed with unit increment.

Output Parameters

<i>y</i>	Contains solution vector x .
----------	--------------------------------

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m, lb
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  REAL alpha
```

```
  REAL val(*)
```

```
  REAL x(*), y(*)
```

```
SUBROUTINE mkl_dbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m, lb
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  DOUBLE PRECISION alpha
```

```
  DOUBLE PRECISION val(*)
```

```
  DOUBLE PRECISION x(*), y(*)
```

```
SUBROUTINE mkl_cbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m, lb
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  COMPLEX alpha
```

```
  COMPLEX val(*)
```

```
  COMPLEX x(*), y(*)
```

```
SUBROUTINE mkl_zbsrsv(transa, m, lb, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m, lb
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  DOUBLE COMPLEX alpha
```

```
  DOUBLE COMPLEX val(*)
```

```
  DOUBLE COMPLEX x(*), y(*)
```

C:

```
void mkl_sbsrsv(char *transa, int *m, int *lb, float *alpha, char *matdescra,
float *val, int *indx, int *pntrb, int *pntre, float *x, float *y);

void mkl_dbsrsv(char *transa, int *m, int *lb, double *alpha, char *matdescra,
double *val, int *indx, int *pntrb, int *pntre, double *x, double *y);

void mkl_cbsrsv(char *transa, int *m, int *lb, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, int *indx, int *pntrb, int *pntre, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zbsrsv(char *transa, int *m, int *lb, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, int *indx, int *pntrb, int *pntre, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?cscsv

Solves a system of linear equations for a sparse matrix in the CSC format.

Syntax**Fortran:**

```
call mkl_scscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_dcscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_ccscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
call mkl_zcscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

C:

```
void mkl_scscsv (char *transa, MKL_INT *m, float *alpha, char *matdescra, float *val,
MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, float *x, float *y);

void mkl_dcscsv (char *transa, MKL_INT *m, double *alpha, char *matdescra, double *val,
MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, double *x, double *y);

void mkl_ccscsv (char *transa, MKL_INT *m, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, MKL_Complex8 *x,
MKL_Complex8 *y);

void mkl_zcscsv (char *transa, MKL_INT *m, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, MKL_Complex16 *x,
MKL_Complex16 *y);
```

Include Files

- **Fortran:** mkl.fi
- **C:** mkl.h

Description

The `mkl_?cscsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSC format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A')* x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports a CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then <i>y</i> := <i>alpha</i> *inv(<i>A</i>)* <i>x</i> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>y</i> := <i>alpha</i> *inv(<i>A'</i>)* <i>x</i> ,
<i>m</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_scscsv. DOUBLE PRECISION for mkl_dcscsv. COMPLEX for mkl_ccscsv. DOUBLE COMPLEX for mkl_zcscsv. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for mkl_scscsv. DOUBLE PRECISION for mkl_dcscsv. COMPLEX for mkl_ccscsv. DOUBLE COMPLEX for mkl_zcscsv. Array containing non-zero elements of the matrix <i>A</i> . For one-based indexing its length is <i>pntre(m)</i> - <i>pntrb(1)</i> . For zero-based indexing its length is <i>pntre(m-1)</i> - <i>pntrb(0)</i> . Refer to <i>values</i> array description in CSC Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx INTEGER. Array containing the row indices for each non-zero element of the matrix *A*. Its length is equal to length of the *val* array.
 Refer to *columns* array description in [CSC Format](#) for more details.

NOTE

Row indices must be sorted in increasing order for each column.

pntrb INTEGER. Array of length *m*.
 For one-based indexing this array contains column indices, such that *pntrb*(*i*) - *pntrb*(1) + 1 is the first index of column *i* in the arrays *val* and *indx*.
 For zero-based indexing this array contains column indices, such that *pntrb*(*i*) - *pntrb*(0) is the first index of column *i* in the arrays *val* and *indx*.
 Refer to *pointerb* array description in [CSC Format](#) for more details.

pntre INTEGER. Array of length *m*.
 For one-based indexing this array contains column indices, such that *pntre*(*i*) - *pntrb*(1) is the last index of column *i* in the arrays *val* and *indx*.
 For zero-based indexing this array contains column indices, such that *pntre*(*i*) - *pntrb*(1) - 1 is the last index of column *i* in the arrays *val* and *indx*.
 Refer to *pointerE* array description in [CSC Format](#) for more details.

x REAL for *mkl_scscsv*.
 DOUBLE PRECISION for *mkl_dcscsv*.
 COMPLEX for *mkl_ccscsv*.
 DOUBLE COMPLEX for *mkl_zcscsv*.
 Array, size at least *m*.
 On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y REAL for *mkl_scscsv*.
 DOUBLE PRECISION for *mkl_dcscsv*.
 COMPLEX for *mkl_ccscsv*.
 DOUBLE COMPLEX for *mkl_zcscsv*.
 Array, size at least *m*.
 On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y Contains the solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  REAL alpha
```

```
  REAL val(*)
```

```
  REAL x(*), y(*)
```

```
SUBROUTINE mkl_dcscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  DOUBLE PRECISION alpha
```

```
  DOUBLE PRECISION val(*)
```

```
  DOUBLE PRECISION x(*), y(*)
```

```
SUBROUTINE mkl_ccscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  COMPLEX alpha
```

```
  COMPLEX val(*)
```

```
  COMPLEX x(*), y(*)
```

```
SUBROUTINE mkl_zcscsv(transa, m, alpha, matdescra, val, indx, pntrb, pntre, x, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m
```

```
  INTEGER indx(*), pntrb(m), pntre(m)
```

```
  DOUBLE COMPLEX alpha
```

```
  DOUBLE COMPLEX val(*)
```

```
  DOUBLE COMPLEX x(*), y(*)
```

C:

```
void mkl_scscsv(char *transa, int *m, float *alpha, char *matdescra,
float *val, int *indx, int *pntrb, int *pntre, float *x, float *y);

void mkl_dcscsv(char *transa, int *m, double *alpha, char *matdescra,
double *val, int *indx, int *pntrb, int *pntre, double *x, double *y);

void mkl_ccscsv(char *transa, int *m, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, int *indx, int *pntrb, int *pntre, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zcscsv(char *transa, int *m, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, int *indx, int *pntrb, int *pntre, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?coosv

Solves a system of linear equations for a sparse matrix in the coordinate format.

Syntax**Fortran:**

```
call mkl_scoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
call mkl_dcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
call mkl_ccoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
call mkl_zcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
```

C:

```
void mkl_scoosv (char *transa, MKL_INT *m, float *alpha, char *matdescra, float *val,
MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, float *x, float *y);
void mkl_dcoosv (char *transa, MKL_INT *m, double *alpha, char *matdescra, double *val,
MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, double *x, double *y);
void mkl_ccoosv (char *transa, MKL_INT *m, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, MKL_Complex8 *x,
MKL_Complex8 *y);
void mkl_zcoosv (char *transa, MKL_INT *m, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, MKL_Complex16 *x,
MKL_Complex16 *y);
```

Include Files

- **Fortran:** mkl.fi
- **C:** mkl.h

Description

The `mkl_?coosv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the coordinate format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A')*x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then <i>y</i> := <i>alpha</i> *inv(<i>A</i>)* <i>x</i> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>y</i> := <i>alpha</i> *inv(<i>A'</i>)* <i>x</i> ,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_scoosv. DOUBLE PRECISION for mkl_dcoosv. COMPLEX for mkl_ccoosv. DOUBLE COMPLEX for mkl_zcoosv. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for mkl_scoosv. DOUBLE PRECISION for mkl_dcoosv. COMPLEX for mkl_ccoosv. DOUBLE COMPLEX for mkl_zcoosv. Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i> , contains the row indices for each non-zero element of the matrix <i>A</i> . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	INTEGER. Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.

nnz INTEGER. Specifies the number of non-zero element of the matrix *A*.

Refer to *nnz* description in [Coordinate Format](#) for more details.

x REAL for mkl_scoosv.

DOUBLE PRECISION for mkl_dcoosv.

COMPLEX for mkl_ccoosv.

DOUBLE COMPLEX for mkl_zcoosv.

Array, size at least *m*.

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y REAL for mkl_scoosv.

DOUBLE PRECISION for mkl_dcoosv.

COMPLEX for mkl_ccoosv.

DOUBLE COMPLEX for mkl_zcoosv.

Array, size at least *m*.

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y Contains solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, nnz
  INTEGER      rowind(*), colind(*)
  REAL         alpha
  REAL         val(*)
  REAL         x(*), y(*)
```

```
SUBROUTINE mkl_dcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1    transa
  CHARACTER      matdescra(*)
  INTEGER        m, nnz
  INTEGER        rowind(*), colind(*)
  DOUBLE PRECISION   alpha
  DOUBLE PRECISION   val(*)
  DOUBLE PRECISION   x(*), y(*)
```

```
SUBROUTINE mkl_ccoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1    transa
  CHARACTER      matdescra(*)
  INTEGER        m, nnz
  INTEGER        rowind(*), colind(*)
  COMPLEX        alpha
  COMPLEX        val(*)
  COMPLEX        x(*), y(*)
```

```
SUBROUTINE mkl_zcoosv(transa, m, alpha, matdescra, val, rowind, colind, nnz, x, y)
```

```
  CHARACTER*1    transa
  CHARACTER      matdescra(*)
  INTEGER        m, nnz
  INTEGER        rowind(*), colind(*)
  DOUBLE COMPLEX   alpha
  DOUBLE COMPLEX   val(*)
  DOUBLE COMPLEX   x(*), y(*)
```

C:

```
void mkl_scoosv(char *transa, int *m, float *alpha, char *matdescra,
float *val, int *rowind, int *colind, int *nnz,
float *x, float *y);

void mkl_dcoosv(char *transa, int *m, double *alpha, char *matdescra,
double *val, int *rowind, int *colind, int *nnz,
double *x, double *y);

void mkl_ccoosv(char *transa, int *m, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, int *rowind, int *colind, int *nnz,
MKL_Complex8 *x, MKL_Complex8 *y);
```

```
void mkl_zcoosv(char *transa, int *m, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, int *rowind, int *colind, int *nnz,
MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?csrmm

Computes matrix - matrix product of a sparse matrix stored in the CSR format.

Syntax

Fortran:

```
call mkl_scsrmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
beta, c, ldc)

call mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
beta, c, ldc)

call mkl_ccsrmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
beta, c, ldc)

call mkl_zcsrmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
beta, c, ldc)
```

C:

```
void mkl_scsrmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *alpha, char
*matdescra, float *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, float *b,
MKL_INT *ldb, float *beta, float *c, MKL_INT *ldc);

void mkl_dcsrmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *alpha, char
*matdescra, double *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, double *b,
MKL_INT *ldb, double *beta, double *c, MKL_INT *ldc);

void mkl_ccsrmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex8 *b, MKL_INT *ldb, MKL_Complex8 *beta, MKL_Complex8 *c, MKL_INT *ldc);

void mkl_zcsrmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16
*alpha, char *matdescra, MKL_Complex16 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT
*pntre, MKL_Complex16 *b, MKL_INT *ldb, MKL_Complex16 *beta, MKL_Complex16 *c, MKL_INT
*ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?csrmm` routine performs a matrix-matrix operation defined as

```
C := alpha*A*B + beta*C
```

or

```
C := alpha*A'*B + beta*C,
```

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in compressed sparse row (CSR) format, *A'* is the transpose of *A*.

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha A^T B + \beta C$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha A^* B + \beta C$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_scsrmm. DOUBLE PRECISION for mkl_dcsrmm. COMPLEX for mkl_ccsrmm. DOUBLE COMPLEX for mkl_zcsrmm. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for mkl_scsrmm. DOUBLE PRECISION for mkl_dcsrmm. COMPLEX for mkl_ccsrmm. DOUBLE COMPLEX for mkl_zcsrmm. Array containing non-zero elements of the matrix <i>A</i> . For one-based indexing its length is <i>pntrc(m)</i> - <i>pntrb(1)</i> . For zero-based indexing its length is <i>pntrc(-1)</i> - <i>pntrb(0)</i> . Refer to <i>values</i> array description in CSR Format for more details.
<i>indx</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i> . Its length is equal to length of the <i>val</i> array. Refer to <i>columns</i> array description in CSR Format for more details.

*pntrb*INTEGER. Array of length m .For one-based indexing this array contains row indices, such that $pntrb(I) - pntrb(1) + 1$ is the first index of row I in the arrays *val* and *indx*.For zero-based indexing this array contains row indices, such that $pntrb(I) - pntrb(0)$ is the first index of row I in the arrays *val* and *indx*.Refer to *pointerb* array description in [CSR Format](#) for more details.*pntre*INTEGER. Array of length m .For one-based indexing this array contains row indices, such that $pntre(I) - pntrb(1)$ is the last index of row I in the arrays *val* and *indx*.For zero-based indexing this array contains row indices, such that $pntre(I) - pntrb(0) - 1$ is the last index of row I in the arrays *val* and *indx*.Refer to *pointerE* array description in [CSR Format](#) for more details.*b*REAL **for** mkl_scsrmm.DOUBLE PRECISION **for** mkl_dcsrmm.COMPLEX **for** mkl_ccsrmm.DOUBLE COMPLEX **for** mkl_zcsrmm.Array, size ldb by at least n for non-transposed matrix A and at least m for transposed for one-based indexing, and (at least k for non-transposed matrix A and at least m for transposed, ldb) for zero-based indexing.On entry with *transa*= 'N' or 'n', the leading k -by- n part of the array *b* must contain the matrix B , otherwise the leading m -by- n part of the array *b* must contain the matrix B .*ldb*INTEGER. Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.*beta*REAL **for** mkl_scsrmm.DOUBLE PRECISION **for** mkl_dcsrmm.COMPLEX **for** mkl_ccsrmm.DOUBLE COMPLEX **for** mkl_zcsrmm.Specifies the scalar *beta*.*c*REAL **for** mkl_scsrmm.DOUBLE PRECISION **for** mkl_dcsrmm.COMPLEX **for** mkl_ccsrmm.DOUBLE COMPLEX **for** mkl_zcsrmm.Array, size ldc by n for one-based indexing, and (m , ldc) for zero-based indexing.On entry, the leading m -by- n part of the array *c* must contain the matrix C , otherwise the leading k -by- n part of the array *c* must contain the matrix C .

ldc INTEGER. Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c Overwritten by the matrix $(\alpha * A * B + \beta * C)$ or $(\alpha * A' * B + \beta * C)$.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
REAL alpha, beta
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
COMPLEX alpha, beta
```

```
COMPLEX val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcsrmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, k, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

DOUBLE COMPLEX alpha, beta

DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_scsrmm(char *transa, int *m, int *n, int *k, float *alpha,
char *matdescra, float *val, int *indx, int *pntrb, int *pntre,
float *b, int *ldb, float *beta, float *c, int *ldc,);

void mkl_dcsrmm(char *transa, int *m, int *n, int *k, double *alpha,
char *matdescra, double *val, int *indx, int *pntrb, int *pntre,
double *b, int *ldb, double *beta, double *c, int *ldc,);

void mkl_ccsrmm(char *transa, int *m, int *n, int *k, double *alpha,
char *matdescra, double *val, int *indx, int *pntrb, int *pntre,
double *b, int *ldb, double *beta, double *c, int *ldc,);

void mkl_zcsrmm(char *transa, int *m, int *n, int *k, double *alpha,
char *matdescra, double *val, int *indx, int *pntrb, int *pntre,
double *b, int *ldb, double *beta, double *c, int *ldc,);
```

[mkl_?bsrmm](#)

Computes matrix - matrix product of a sparse matrix stored in the BSR format.

Syntax**Fortran:**

```
call mkl_sbsrmm(transa, m, n, k, lb, alpha, matdescra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc)

call mkl_dbsrmm(transa, m, n, k, lb, alpha, matdescra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc)

call mkl_cbsrmm(transa, m, n, k, lb, alpha, matdescra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc)

call mkl_zbsrmm(transa, m, n, k, lb, alpha, matdescra, val, indx, pntrb, pntre, b,
ldb, beta, c, ldc)
```

C:

```

void mkl_sbsrmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_INT *lb, float
*alpha, char *matdescra, float *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
float *b, MKL_INT *ldb, float *beta, float *c, MKL_INT *ldc);

void mkl_dbsrmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_INT *lb, double
*alpha, char *matdescra, double *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
double *b, MKL_INT *ldb, double *beta, double *c, MKL_INT *ldc);

void mkl_cbsrmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_INT *lb,
MKL_Complex8 *alpha, char *matdescra, MKL_Complex8 *val, MKL_INT *indx, MKL_INT
*pntrb, MKL_INT *pntre, MKL_Complex8 *b, MKL_INT *ldb, MKL_Complex8 *beta,
MKL_Complex8 *c, MKL_INT *ldc);

void mkl_zbsrmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_INT *lb,
MKL_Complex16 *alpha, char *matdescra, MKL_Complex16 *val, MKL_INT *indx, MKL_INT
*pntrb, MKL_INT *pntre, MKL_Complex16 *b, MKL_INT *ldb, MKL_Complex16 *beta,
MKL_Complex16 *c, MKL_INT *ldc);

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?bsrmm` routine performs a matrix-matrix operation defined as

$$C := \alpha A^* B + \beta C$$

or

$$C := \alpha A'^* B + \beta C,$$

where:

`alpha` and `beta` are scalars,

`B` and `C` are dense matrices, `A` is an m -by- k sparse matrix in block sparse row (BSR) format, `A'` is the transpose of `A`.

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>transa</code>	CHARACTER*1. Specifies the operation. If <code>transa = 'N'</code> or <code>'n'</code> , then the matrix-matrix product is computed as $C := \alpha A^* B + \beta C$ If <code>transa = 'T'</code> or <code>'t'</code> or <code>'C'</code> or <code>'c'</code> , then the matrix-vector product is computed as $C := \alpha A'^* B + \beta C,$
<code>m</code>	INTEGER. Number of block rows of the matrix <code>A</code> .
<code>n</code>	INTEGER. Number of columns of the matrix <code>C</code> .

<i>k</i>	INTEGER. Number of block columns of the matrix <i>A</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>alpha</i>	<p>REAL for mkl_sbsrmm.</p> <p>DOUBLE PRECISION for mkl_dbsrmm.</p> <p>COMPLEX for mkl_cbsrmm.</p> <p>DOUBLE COMPLEX for mkl_zbsrmm.</p> <p>Specifies the scalar <i>alpha</i>.</p>
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	<p>REAL for mkl_sbsrmm.</p> <p>DOUBLE PRECISION for mkl_dbsrmm.</p> <p>COMPLEX for mkl_cbsrmm.</p> <p>DOUBLE COMPLEX for mkl_zbsrmm.</p> <p>Array containing elements of non-zero blocks of the matrix <i>A</i>. Its length is equal to the number of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb</i>*<i>lb</i>. Refer to the <i>values</i> array description in BSR Format for more details.</p>
<i>indx</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> . Refer to the <i>columns</i> array description in BSR Format for more details.
<i>pntrb</i>	<p>INTEGER. Array of length <i>m</i>.</p> <p>For one-based indexing: this array contains row indices, such that <i>pntrb</i>(<i>I</i>) - <i>pntrb</i>(1) + 1 is the first index of block row <i>I</i> in the array <i>indx</i>.</p> <p>For zero-based indexing: this array contains row indices, such that <i>pntrb</i>(<i>I</i>) - <i>pntrb</i>(0) is the first index of block row <i>I</i> in the array <i>indx</i>.</p> <p>Refer to <i>pointerB</i> array description in BSR Format for more details.</p>
<i>pntre</i>	<p>INTEGER. Array of length <i>m</i>.</p> <p>For one-based indexing this array contains row indices, such that <i>pntre</i>(<i>I</i>) - <i>pntrb</i>(1) is the last index of block row <i>I</i> in the array <i>indx</i>.</p> <p>For zero-based indexing this array contains row indices, such that <i>pntre</i>(<i>I</i>) - <i>pntrb</i>(0) - 1 is the last index of block row <i>I</i> in the array <i>indx</i>.</p> <p>Refer to <i>pointerE</i> array description in BSR Format for more details.</p>
<i>b</i>	<p>REAL for mkl_sbsrmm.</p> <p>DOUBLE PRECISION for mkl_dbsrmm.</p> <p>COMPLEX for mkl_cbsrmm.</p>

DOUBLE COMPLEX for `mkl_zbsrmm`.

Array, size l_{db} by at least n for non-transposed matrix A and at least m for transposed for one-based indexing, and (at least k for non-transposed matrix A and at least m for transposed, l_{db}) for zero-based indexing.

On entry with $\text{transa} = \text{'N'}$ or ' n ', the leading n -by- k block part of the array b must contain the matrix B , otherwise the leading m -by- n block part of the array b must contain the matrix B .

l_{db} INTEGER. Specifies the leading dimension (in blocks) of b as declared in the calling (sub)program.

β REAL for `mkl_sbsrmm`.

DOUBLE PRECISION for `mkl_dbsrmm`.

COMPLEX for `mkl_cbsrmm`.

DOUBLE COMPLEX for `mkl_zbsrmm`.

Specifies the scalar β .

c REAL for `mkl_sbsrmm`.

DOUBLE PRECISION for `mkl_dbsrmm`.

COMPLEX for `mkl_cbsrmm`.

DOUBLE COMPLEX for `mkl_zbsrmm`.

Array, size (l_{dc}, n) for one-based indexing, size (k, l_{dc}) for zero-based indexing.

On entry, the leading m -by- n block part of the array c must contain the matrix C , otherwise the leading n -by- k block part of the array c must contain the matrix C .

l_{dc} INTEGER. Specifies the leading dimension (in blocks) of c as declared in the calling (sub)program.

Output Parameters

c Overwritten by the matrix $(\alpha * A * B + \beta * C)$ or $(\alpha * A' * B + \beta * C)$.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrmm(transa, m, n, k, lb, alpha, matdescra, val,  
          indx, pntrb, pntre, b, ldb, beta, c, ldc)
```

```
  CHARACTER*1   transa
```

```
  CHARACTER     matdescra(*)
```

```
  INTEGER       m, n, k, ld, ldb, ldc
```

```
  INTEGER       indx(*), pntrb(m), pntre(m)
```

```
  REAL          alpha, beta
```

```
  REAL          val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dbsrmm(transa, m, n, k, lb, alpha, matdescra, val,  
          indx, pntrb, pntre, b, ldb, beta, c, ldc)
```

```
  CHARACTER*1   transa
```

```
  CHARACTER     matdescra(*)
```

```
  INTEGER       m, n, k, ld, ldb, ldc
```

```
  INTEGER       indx(*), pntrb(m), pntre(m)
```

```
  DOUBLE PRECISION      alpha, beta
```

```
  DOUBLE PRECISION      val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_cbsrmm(transa, m, n, k, lb, alpha, matdescra, val,  
          indx, pntrb, pntre, b, ldb, beta, c, ldc)
```

```
  CHARACTER*1   transa
```

```
  CHARACTER     matdescra(*)
```

```
  INTEGER       m, n, k, ld, ldb, ldc
```

```
  INTEGER       indx(*), pntrb(m), pntre(m)
```

```
  COMPLEX      alpha, beta
```

```
  COMPLEX      val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zbsrmm(transa, m, n, k, lb, alpha, matdescra, val,  
          indx, pntrb, pntre, b, ldb, beta, c, ldc)
```

```
  CHARACTER*1   transa
```

```
  CHARACTER     matdescra(*)
```

```
  INTEGER       m, n, k, ld, ldb, ldc
```

```
  INTEGER       indx(*), pntrb(m), pntre(m)
```

```
  DOUBLE COMPLEX      alpha, beta
```

```
  DOUBLE COMPLEX      val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_sbsrmm(char *transa, int *m, int *n, int *k, int *lb,
float *alpha, char *matdescra, float *val, int *indx, int *pntrb,
int *pntre, float *b, int *ldb, float *beta, float *c, int *ldc,);

void mkl_dbsrmm(char *transa, int *m, int *n, int *k, int *lb,
double *alpha, char *matdescra, double *val, int *indx, int *pntrb,
int *pntre, double *b, int *ldb, double *beta, double *c, int *ldc,);

void mkl_cbsrmm(char *transa, int *m, int *n, int *k, int *lb,
MKL_Complex8 *alpha, char *matdescra, MKL_Complex8 *val, int *indx, int *pntrb,
int *pntre, MKL_Complex8 *b, int *ldb, MKL_Complex8 *beta, MKL_Complex8 *c, int *ldc,);

void mkl_zbsrmm(char *transa, int *m, int *n, int *k, int *lb,
MKL_Complex16 *alpha, char *matdescra, MKL_Complex16 *val, int *indx, int *pntrb,
int *pntre, MKL_Complex16 *b, int *ldb, MKL_Complex16 *beta, MKL_Complex16 *c, int *ldc,);
```

mkl_?cscmm

Computes matrix-matrix product of a sparse matrix stored in the CSC format.

Syntax**Fortran:**

```
call mkl_scscmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
beta, c, ldc)

call mkl_dcscmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
beta, c, ldc)

call mkl_ccscmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
beta, c, ldc)

call mkl_zcscmm(transa, m, n, k, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
beta, c, ldc)
```

C:

```
void mkl_scscmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *alpha, char
*matdescra, float *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, float *b,
MKL_INT *ldb, float *beta, float *c, MKL_INT *ldc);

void mkl_dcscmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *alpha, char
*matdescra, double *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, double *b,
MKL_INT *ldb, double *beta, double *c, MKL_INT *ldc);

void mkl_ccscmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex8 *b, MKL_INT *ldb, MKL_Complex8 *beta, MKL_Complex8 *c, MKL_INT *ldc);

void mkl_zcscmm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16
*alpha, char *matdescra, MKL_Complex16 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT
*pntre, MKL_Complex16 *b, MKL_INT *ldb, MKL_Complex16 *beta, MKL_Complex16 *c, MKL_INT
*ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?cscmm` routine performs a matrix-matrix operation defined as

```
C := alpha*A*B + beta*C
```

or

```
C := alpha*A'*B + beta*C,
```

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in compressed sparse column (CSC) format, *A'* is the transpose of *A*.

NOTE

This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha A^* B + \beta C$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha A'^* B + \beta C$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_scscmm</code> . DOUBLE PRECISION for <code>mkl_dcscmm</code> . COMPLEX for <code>mkl_ccscmm</code> . DOUBLE COMPLEX for <code>mkl_zcscmm</code> . Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for <code>mkl_scscmm</code> .

DOUBLE PRECISION **for** mkl_dcscmm.

COMPLEX **for** mkl_ccscmm.

DOUBLE COMPLEX **for** mkl_zcscmm.

Array containing non-zero elements of the matrix A .

For one-based indexing its length is $pntrb(k) - pntrb(1)$.

For zero-based indexing its length is $pntrb(m-1) - pntrb(0)$.

Refer to *values* array description in [CSC Format](#) for more details.

indx

INTEGER. Array containing the row indices for each non-zero element of the matrix A . Its length is equal to length of the *val* array.

Refer to *rows* array description in [CSC Format](#) for more details.

pntrb

INTEGER. Array of length k .

For one-based indexing this array contains column indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of column i in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that $pntrb(i) - pntrb(0)$ is the first index of column i in the arrays *val* and *indx*.

Refer to *pointerb* array description in [CSC Format](#) for more details.

pntre

INTEGER. Array of length k .

For one-based indexing this array contains column indices, such that $pntre(i) - pntrb(1)$ is the last index of column i in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that $pntre(i) - pntrb(1) - 1$ is the last index of column i in the arrays *val* and *indx*.

Refer to *pointerE* array description in [CSC Format](#) for more details.

b

REAL **for** mkl_scscmm.

DOUBLE PRECISION **for** mkl_dcscmm.

COMPLEX **for** mkl_ccscmm.

DOUBLE COMPLEX **for** mkl_zcscmm.

Array, size ldb by at least n for non-transposed matrix A and at least m for transposed for one-based indexing, and (at least k for non-transposed matrix A and at least m for transposed, ldb) for zero-based indexing.

On entry with $transa = 'N'$ or ' n ', the leading k -by- n part of the array *b* must contain the matrix B , otherwise the leading m -by- n part of the array *b* must contain the matrix B .

ldb

INTEGER. Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

<i>beta</i>	REAL*8. Specifies the scalar <i>beta</i> .
<i>c</i>	REAL for mkl_scscmm. DOUBLE PRECISION for mkl_dcscmm. COMPLEX for mkl_ccscmm. DOUBLE COMPLEX for mkl_zcscmm.
	Array, size <i>lrc</i> by <i>n</i> for one-based indexing, and (<i>m</i> , <i>lrc</i>) for zero-based indexing.
	On entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> , otherwise the leading <i>k</i> -by- <i>n</i> part of the array <i>c</i> must contain the matrix <i>C</i> .
<i>lrc</i>	INTEGER. Specifies the leading dimension of <i>c</i> for one-based indexing, and the second dimension of <i>c</i> for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

<i>c</i>	Overwritten by the matrix (<i>alpha</i> * <i>A</i> * <i>B</i> + <i>beta</i> * <i>C</i>) or (<i>alpha</i> * <i>A'</i> * <i>B</i> + <i>beta</i> * <i>C</i>).
----------	--

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scscmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER indx(*), pntrb(k), pntre(k)
```

```
REAL alpha, beta
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcscmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc
```

```
INTEGER indx(*), pntrb(k), pntre(k)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```

SUBROUTINE mkl_ccscmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, n, k, ldb, ldc
  INTEGER indx(*), pntrb(k), pntre(k)
  COMPLEX alpha, beta
  COMPLEX val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zcscmm(transa, m, n, k, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, beta, c, ldc)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, n, k, ldb, ldc
  INTEGER indx(*), pntrb(k), pntre(k)
  DOUBLE COMPLEX alpha, beta
  DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)

```

C:

```

void mkl_scscmm(char *transa, int *m, int *n, int *k,
float *alpha, char *matdescra, float *val, int *indx,
int *pntrb, int *pntre, float *b, int *ldb,
float *beta, float *c, int *ldc);

void mkl_dcscmm(char *transa, int *m, int *n, int *k,
double *alpha, char *matdescra, double *val, int *indx,
int *pntrb, int *pntre, double *b, int *ldb,
double *beta, double *c, int *ldc);

void mkl_ccscmm(char *transa, int *m, int *n, int *k,
MKL_Complex8 *alpha, char *matdescra, MKL_Complex8 *val, int *indx,
int *pntrb, int *pntre, MKL_Complex8 *b, int *ldb,
MKL_Complex8 *beta, MKL_Complex8 *c, int *ldc);

void mkl_zcscmm(char *transa, int *m, int *n, int *k,
MKL_Complex16 *alpha, char *matdescra, MKL_Complex16 *val, int *indx,
int *pntrb, int *pntre, MKL_Complex16 *b, int *ldb,
MKL_Complex16 *beta, MKL_Complex16 *c, int *ldc);

```

mkl_?coomm

Computes matrix-matrix product of a sparse matrix stored in the coordinate format.

Syntax

Fortran:

```
call mkl_scoomm(transa, m, n, k, alpha, matdescra, val, rowind, colind, nnz, b, ldb,
beta, c, ldc)

call mkl_dcoomm(transa, m, n, k, alpha, matdescra, val, rowind, colind, nnz, b, ldb,
beta, c, ldc)

call mkl_ccoomm(transa, m, n, k, alpha, matdescra, val, rowind, colind, nnz, b, ldb,
beta, c, ldc)

call mkl_zcoomm(transa, m, n, k, alpha, matdescra, val, rowind, colind, nnz, b, ldb,
beta, c, ldc)
```

C:

```
void mkl_scoomm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *alpha, char
*matdescra, float *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, float *b,
MKL_INT *ldb, float *beta, float *c, MKL_INT *ldc);

void mkl_dcoomm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *alpha, char
*matdescra, double *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, double *b,
MKL_INT *ldb, double *beta, double *c, MKL_INT *ldc);

void mkl_ccoomm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz,
MKL_Complex8 *b, MKL_INT *ldb, MKL_Complex8 *beta, MKL_Complex8 *c, MKL_INT *ldc);

void mkl_zcoomm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16
*alpha, char *matdescra, MKL_Complex16 *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT
*nnz, MKL_Complex16 *b, MKL_INT *ldb, MKL_Complex16 *beta, MKL_Complex16 *c, MKL_INT
*ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?coomm` routine performs a matrix-matrix operation defined as

```
C := alpha*A*B + beta*C
```

or

```
C := alpha*A'*B + beta*C,
```

where:

`alpha` and `beta` are scalars,

`B` and `C` are dense matrices, `A` is an m -by- k sparse matrix in the coordinate format, A' is the transpose of A .

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha A^T B + \beta C$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha A^H B + \beta C$,
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>n</i>	INTEGER. Number of columns of the matrix C .
<i>k</i>	INTEGER. Number of columns of the matrix A .
<i>alpha</i>	REAL for mkl_scoomm. DOUBLE PRECISION for mkl_dcoomm. COMPLEX for mkl_ccoomm. DOUBLE COMPLEX for mkl_zcoomm. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for mkl_scoomm. DOUBLE PRECISION for mkl_dcoomm. COMPLEX for mkl_ccoomm. DOUBLE COMPLEX for mkl_zcoomm. Array of length <i>nnz</i> , contains non-zero elements of the matrix A in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i> , contains the row indices for each non-zero element of the matrix A . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	INTEGER. Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix A . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix A . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>b</i>	REAL for mkl_scoomm.

DOUBLE PRECISION for mkl_dcoomm.

COMPLEX for mkl_ccoomm.

DOUBLE COMPLEX for mkl_zcoomm.

Array, size ldb by at least n for non-transposed matrix A and at least m for transposed for one-based indexing, and (at least k for non-transposed matrix A and at least m for transposed, ldb) for zero-based indexing.

On entry with $transa = 'N'$ or ' n ', the leading k -by- n part of the array b must contain the matrix B , otherwise the leading m -by- n part of the array b must contain the matrix B .

ldb

INTEGER. Specifies the leading dimension of b for one-based indexing, and the second dimension of b for zero-based indexing, as declared in the calling (sub)program.

β

REAL for mkl_scoomm.

DOUBLE PRECISION for mkl_dcoomm.

COMPLEX for mkl_ccoomm.

DOUBLE COMPLEX for mkl_zcoomm.

Specifies the scalar β .

c

REAL for mkl_scoomm.

DOUBLE PRECISION for mkl_dcoomm.

COMPLEX for mkl_ccoomm.

DOUBLE COMPLEX for mkl_zcoomm.

Array, size ldc by n for one-based indexing, and (m , ldc) for zero-based indexing.

On entry, the leading m -by- n part of the array c must contain the matrix C , otherwise the leading k -by- n part of the array c must contain the matrix C .

ldc

INTEGER. Specifies the leading dimension of c for one-based indexing, and the second dimension of c for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c

Overwritten by the matrix $(\alpha * A * B + \beta * C)$ or $(\alpha * A' * B + \beta * C)$.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoomm(transa, m, n, k, alpha, matdescra, val,
    rowind, colind, nnz, b, ldb, beta, c, ldc)
    CHARACTER*1    transa
    CHARACTER      matdescra(*)
    INTEGER        m, n, k, ldb, ldc, nnz
    INTEGER        rowind(*), colind(*)
    REAL           alpha, beta
    REAL           val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_dcoomm(transa, m, n, k, alpha, matdescra, val,
    rowind, colind, nnz, b, ldb, beta, c, ldc)
    CHARACTER*1    transa
    CHARACTER      matdescra(*)
    INTEGER        m, n, k, ldb, ldc, nnz
    INTEGER        rowind(*), colind(*)
    DOUBLE PRECISION   alpha, beta
    DOUBLE PRECISION   val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_ccoomm(transa, m, n, k, alpha, matdescra, val,
    rowind, colind, nnz, b, ldb, beta, c, ldc)
    CHARACTER*1    transa
    CHARACTER      matdescra(*)
    INTEGER        m, n, k, ldb, ldc, nnz
    INTEGER        rowind(*), colind(*)
    COMPLEX        alpha, beta
    COMPLEX        val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zcoomm(transa, m, n, k, alpha, matdescra, val,
    rowind, colind, nnz, b, ldb, beta, c, ldc)
    CHARACTER*1    transa
    CHARACTER      matdescra(*)
    INTEGER        m, n, k, ldb, ldc, nnz
    INTEGER        rowind(*), colind(*)
    DOUBLE COMPLEX   alpha, beta
    DOUBLE COMPLEX   val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_scoomm(char *transa, int *m, int *n, int *k, float *alpha,
char *matdescra, float *val, int *rowind, int *colind, int *nnz,
float *b, int *ldb, float *beta, float *c, int *ldc);

void mkl_dcoomm(char *transa, int *m, int *n, int *k, double *alpha,
char *matdescra, double *val, int *rowind, int *colind, int *nnz,
double *b, int *ldb, double *beta, double *c, int *ldc);

void mkl_ccoomm(char *transa, int *m, int *n, int *k, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, int *rowind, int *colind, int *nnz,
MKL_Complex8 *b, int *ldb, MKL_Complex8 *beta, MKL_Complex8 *c, int *ldc);

void mkl_zcoomm(char *transa, int *m, int *n, int *k, MKL_Complex16 *alpha,
char *matdescra, MKL_Complex16 *val, int *rowind, int *colind, int *nnz,
MKL_Complex16 *b, int *ldb, MKL_Complex16 *beta, MKL_Complex16 *c, int *ldc);
```

mkl_?csrsm

Solves a system of linear matrix equations for a sparse matrix in the CSR format.

Syntax**Fortran:**

```
call mkl_scsrsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)

call mkl_dcsrsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)

call mkl_ccsrsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)

call mkl_zcsrsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)
```

C:

```
void mkl_scsrsm (char *transa, MKL_INT *m, MKL_INT *n, float *alpha, char *matdescra,
float *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, float *b, MKL_INT *ldb,
float *c, MKL_INT *ldc);

void mkl_dcsrsm (char *transa, MKL_INT *m, MKL_INT *n, double *alpha, char *matdescra,
double *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, double *b, MKL_INT *ldb,
double *c, MKL_INT *ldc);

void mkl_ccsrsm (char *transa, MKL_INT *m, MKL_INT *n, MKL_Complex8 *alpha, char
*matdescra, MKL_Complex8 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex8 *b, MKL_INT *ldb, MKL_Complex8 *c, MKL_INT *ldc);

void mkl_zcsrsm (char *transa, MKL_INT *m, MKL_INT *n, MKL_Complex16 *alpha, char
*matdescra, MKL_Complex16 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex16 *b, MKL_INT *ldb, MKL_Complex16 *c, MKL_INT *ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?csrsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSR format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(A')*B,
```

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then <i>C</i> := <i>alpha</i> *inv(<i>A</i>)* <i>B</i> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>C</i> := <i>alpha</i> *inv(<i>A'</i>)* <i>B</i> ,
<i>m</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	REAL for <code>mkl_scsrsm</code> . DOUBLE PRECISION for <code>mkl_dcsrsm</code> . COMPLEX for <code>mkl_ccsrsm</code> . DOUBLE COMPLEX for <code>mkl_zcsrsm</code> . Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for <code>mkl_scsrsm</code> . DOUBLE PRECISION for <code>mkl_dcsrsm</code> . COMPLEX for <code>mkl_ccsrsm</code> .

DOUBLE COMPLEX for `mkl_zcsrsm`.

Array containing non-zero elements of the matrix A .

For one-based indexing its length is $pntre(m) - pntrb(1)$.

For zero-based indexing its length is $pntre(m-1) - pntrb(0)$.

Refer to `values` array description in [CSR Format](#) for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx

INTEGER. Array containing the column indices for each non-zero element of the matrix A . Its length is equal to length of the `val` array.

Refer to `columns` array description in [CSR Format](#) for more details.

NOTE

Column indices must be sorted in increasing order for each row.

pntrb

INTEGER. Array of length m .

For one-based indexing this array contains row indices, such that $pntrb(i) - pntrb(1) + 1$ is the first index of row i in the arrays `val` and `indx`.

For zero-based indexing this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of row i in the arrays `val` and `indx`. Refer to `pointerb` array description in [CSR Format](#) for more details.

pntre

INTEGER. Array of length m .

For one-based indexing this array contains row indices, such that $pntre(i) - pntrb(1)$ is the last index of row i in the arrays `val` and `indx`.

For zero-based indexing this array contains row indices, such that $pntre(i) - pntrb(0) - 1$ is the last index of row i in the arrays `val` and `indx`. Refer to `pointerE` array description in [CSR Format](#) for more details.

b

REAL for `mkl_scsrsm`.

DOUBLE PRECISION for `mkl_dcsrsm`.

COMPLEX for `mkl_ccsrsm`.

DOUBLE COMPLEX for `mkl_zcsrsm`.

Array, size (l_{db}, n) for one-based indexing, and (m, l_{db}) for zero-based indexing.

On entry the leading m -by- n part of the array *b* must contain the matrix B .

ldb

INTEGER. Specifies the leading dimension of *b* for one-based indexing, and the second dimension of *b* for zero-based indexing, as declared in the calling (sub)program.

ldc INTEGER. Specifies the leading dimension of *c* for one-based indexing, and the second dimension of *c* for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c REAL*8.
Array, size *ldc* by *n* for one-based indexing, and (*m*, *ldc*) for zero-based indexing.
The leading *m*-by-*n* part of the array *c* contains the output matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsm(transa, m, n, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
REAL alpha
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcsrsm(transa, m, n, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
DOUBLE PRECISION alpha
```

```
DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_ccsrsm(transa, m, n, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntre(m)
```

```
COMPLEX alpha
```

```
COMPLEX val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zcsrsm(transa, m, n, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

DOUBLE COMPLEX alpha

DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_scsrsm(char *transa, int *m, int *n, float *alpha,
char *matdescra, float *val, int *indx, int *pntrb,
int *pntre, float *b, int *ldb, float *c, int *ldc);

void mkl_dcsrsm(char *transa, int *m, int *n, double *alpha,
char *matdescra, double *val, int *indx, int *pntrb,
int *pntre, double *b, int *ldb, double *c, int *ldc);

void mkl_ccsrsm(char *transa, int *m, int *n, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, int *indx, int *pntrb,
int *pntre, MKL_Complex8 *b, int *ldb, MKL_Complex8 *c, int *ldc);

void mkl_zcsrsm(char *transa, int *m, int *n, MKL_Complex16 *alpha,
char *matdescra, MKL_Complex16 *val, int *indx, int *pntrb,
int *pntre, MKL_Complex16 *b, int *ldb, MKL_Complex16 *c, int *ldc);
```

mkl_?cscsm

Solves a system of linear matrix equations for a sparse matrix in the CSC format.

Syntax**Fortran:**

```
call mkl_scscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)

call mkl_dcscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)

call mkl_ccscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)

call mkl_zcscsm(transa, m, n, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c,
ldc)
```

C:

```

void mkl_scscsm (char *transa, MKL_INT *m, MKL_INT *n, float *alpha, char *matdescra,
float *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, float *b, MKL_INT *ldb,
float *c, MKL_INT *ldc);

void mkl_dcscsm (char *transa, MKL_INT *m, MKL_INT *n, double *alpha, char *matdescra,
double *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, double *b, MKL_INT *ldb,
double *c, MKL_INT *ldc);

void mkl_ccscsm (char *transa, MKL_INT *m, MKL_INT *n, MKL_Complex8 *alpha, char
*matdescra, MKL_Complex8 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex8 *b, MKL_INT *ldb, MKL_Complex8 *c, MKL_INT *ldc);

void mkl_zcscsm (char *transa, MKL_INT *m, MKL_INT *n, MKL_Complex16 *alpha, char
*matdescra, MKL_Complex16 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre,
MKL_Complex16 *b, MKL_INT *ldb, MKL_Complex16 *c, MKL_INT *ldc);

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?cscsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSC format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(A')*B,
```

where:

`alpha` is scalar, `B` and `C` are dense matrices, `A` is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, `A'` is the transpose of `A`.

NOTE

This routine supports a CSC format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>transa</code>	CHARACTER*1. Specifies the system of equations. If <code>transa = 'N'</code> or <code>'n'</code> , then <code>C := alpha*inv(A)*B</code> If <code>transa = 'T'</code> or <code>'t'</code> or <code>'C'</code> or <code>'c'</code> , then <code>C := alpha*inv(A')*B,</code>
<code>m</code>	INTEGER. Number of columns of the matrix <code>A</code> .
<code>n</code>	INTEGER. Number of columns of the matrix <code>C</code> .
<code>alpha</code>	REAL for <code>mkl_scscsm</code> . DOUBLE PRECISION for <code>mkl_dcscsm</code> . COMPLEX for <code>mkl_ccscsm</code> .

DOUBLE COMPLEX for `mkl_zcscsm`.

Specifies the scalar *alpha*.

matdescra

CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in [Table "Possible Values of the Parameter *matdescra* \(*descra*\)"](#). Possible combinations of element values of this parameter are given in [Table "Possible Combinations of Element Values of the Parameter *matdescra*"](#).

val

REAL for `mkl_scscsm`.

DOUBLE PRECISION for `mkl_dcscsm`.

COMPLEX for `mkl_ccscsm`.

DOUBLE COMPLEX for `mkl_zcscsm`.

Array containing non-zero elements of the matrix *A*.

For one-based indexing its length is $pntre(k) - pntrb(1)$.

For zero-based indexing its length is $pntre(m-1) - pntrb(0)$.

Refer to *values* array description in [CSC Format](#) for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx

INTEGER. Array containing the row indices for each non-zero element of the matrix *A*. Its length is equal to length of the *val* array.

Refer to *rows* array description in [CSC Format](#) for more details.

NOTE

Row indices must be sorted in increasing order for each column.

pntrb

INTEGER. Array of length *m*.

For one-based indexing this array contains column indices, such that $pntrb(I) - pntrb(1) + 1$ is the first index of column *I* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that $pntrb(I) - pntrb(0)$ is the first index of column *I* in the arrays *val* and *indx*.

Refer to *pointerb* array description in [CSC Format](#) for more details.

pntre

INTEGER. Array of length *m*.

For one-based indexing this array contains column indices, such that $pntre(I) - pntre(1)$ is the last index of column *I* in the arrays *val* and *indx*.

For zero-based indexing this array contains column indices, such that $pntre(I) - pntrb(1)-1$ is the last index of column I in the arrays `val` and `indx`.

Refer to `pointerE` array description in [CSC Format](#) for more details.

b

REAL for `mkl_scscsm`.

DOUBLE PRECISION for `mkl_dcscsm`.

COMPLEX for `mkl_ccscsm`.

DOUBLE COMPLEX for `mkl_zcscsm`.

Array, size ldb by n for one-based indexing, and (m, ldb) for zero-based indexing.

On entry the leading m -by- n part of the array b must contain the matrix B .

ldb

INTEGER. Specifies the leading dimension of b for one-based indexing, and the second dimension of b for zero-based indexing, as declared in the calling (sub)program.

ldc

INTEGER. Specifies the leading dimension of c for one-based indexing, and the second dimension of c for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

c

REAL for `mkl_scscsm`.

DOUBLE PRECISION for `mkl_dcscsm`.

COMPLEX for `mkl_ccscsm`.

DOUBLE COMPLEX for `mkl_zcscsm`.

Array, size ldc by n for one-based indexing, and (m, ldc) for zero-based indexing.

The leading m -by- n part of the array c contains the output matrix C .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scscsm(transa, m, n, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, c, ldc)

CHARACTER*1 transa
CHARACTER matdescra(*)
INTEGER m, n, ldb, ldc
INTEGER indx(*), pntrb(m), pntre(m)
REAL alpha
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcscsm(transa, m, n, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

DOUBLE PRECISION alpha

DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_ccscsm(transa, m, n, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

COMPLEX alpha

COMPLEX val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zcscsm(transa, m, n, alpha, matdescra, val, indx,
pntrb, pntre, b, ldb, c, ldc)

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, ldb, ldc

INTEGER indx(*), pntrb(m), pntre(m)

DOUBLE COMPLEX alpha

DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_scscsm(char *transa, int *m, int *n, float *alpha,
char *matdescra, float *val, int *indx, int *pntrb,
int *pntre, float *b, int *ldb, float *c, int *ldc);

void mkl_dcscsm(char *transa, int *m, int *n, double *alpha,
char *matdescra, double *val, int *indx, int *pntrb,
int *pntre, double *b, int *ldb, double *c, int *ldc);

void mkl_ccscsm(char *transa, int *m, int *n, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, int *indx, int *pntrb,
int *pntre, MKL_Complex8 *b, int *ldb, MKL_Complex8 *c, int *ldc);
```

```
void mkl_zcscsm(char *transa, int *m, int *n, MKL_Complex16 *alpha,
char *matdescra, MKL_Complex16 *val, int *indx, int *pntrb,
int *pntre, MKL_Complex16 *b, int *ldb, MKL_Complex16 *c, int *ldc);
```

mkl_?coasm

Solves a system of linear matrix equations for a sparse matrix in the coordinate format.

Syntax

Fortran:

```
call mkl_scoasm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c,
ldc)
call mkl_dcoasm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c,
ldc)
call mkl_ccoasm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c,
ldc)
call mkl_zcoasm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c,
ldc)
```

C:

```
void mkl_scoasm (char *transa, MKL_INT *m, MKL_INT *n, float *alpha, char *matdescra,
float *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, float *b, MKL_INT *ldb,
float *c, MKL_INT *ldc);
void mkl_dcoasm (char *transa, MKL_INT *m, MKL_INT *n, double *alpha, char *matdescra,
double *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz, double *b, MKL_INT *ldb,
double *c, MKL_INT *ldc);
void mkl_ccoasm (char *transa, MKL_INT *m, MKL_INT *n, MKL_Complex8 *alpha, char
*matdescra, MKL_Complex8 *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz,
MKL_Complex8 *b, MKL_INT *ldb, MKL_Complex8 *c, MKL_INT *ldc);
void mkl_zcoasm (char *transa, MKL_INT *m, MKL_INT *n, MKL_Complex16 *alpha, char
*matdescra, MKL_Complex16 *val, MKL_INT *rowind, MKL_INT *colind, MKL_INT *nnz,
MKL_Complex16 *b, MKL_INT *ldb, MKL_Complex16 *c, MKL_INT *ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?coasm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the coordinate format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(A')*B,
```

where:

`alpha` is scalar, `B` and `C` are dense matrices, `A` is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, `A'` is the transpose of `A`.

NOTE

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then the matrix-matrix product is computed as $C := \alpha * \text{inv}(A) * B$ If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $C := \alpha * \text{inv}(A') * B$,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	REAL for mkl_scoosm. DOUBLE PRECISION for mkl_dcoosm. COMPLEX for mkl_ccoosm. DOUBLE COMPLEX for mkl_zcoosm. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for mkl_scoosm. DOUBLE PRECISION for mkl_dcoosm. COMPLEX for mkl_ccoosm. DOUBLE COMPLEX for mkl_zcoosm. Array of length <i>nnz</i> , contains non-zero elements of the matrix <i>A</i> in the arbitrary order. Refer to <i>values</i> array description in Coordinate Format for more details.
<i>rowind</i>	INTEGER. Array of length <i>nnz</i> , contains the row indices for each non-zero element of the matrix <i>A</i> . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	INTEGER. Array of length <i>nnz</i> , contains the column indices for each non-zero element of the matrix <i>A</i> . Refer to <i>columns</i> array description in Coordinate Format for more details.
<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix <i>A</i> .

Refer to *nnz* description in [Coordinate Format](#) for more details.

<i>b</i>	REAL for mkl_scoosm. DOUBLE PRECISION for mkl_dcoosm. COMPLEX for mkl_ccoosm. DOUBLE COMPLEX for mkl_zcoosm. Array, size <i>ldb</i> by <i>n</i> for one-based indexing, and (<i>m</i> , <i>ldb</i>) for zero-based indexing. Before entry the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>ldb</i>	INTEGER. Specifies the leading dimension of <i>b</i> for one-based indexing, and the second dimension of <i>b</i> for zero-based indexing, as declared in the calling (sub)program.
<i>ldc</i>	INTEGER. Specifies the leading dimension of <i>c</i> for one-based indexing, and the second dimension of <i>c</i> for zero-based indexing, as declared in the calling (sub)program.

Output Parameters

<i>c</i>	REAL for mkl_scoosm. DOUBLE PRECISION for mkl_dcoosm. COMPLEX for mkl_ccoosm. DOUBLE COMPLEX for mkl_zcoosm. Array, size <i>ldc</i> by <i>n</i> for one-based indexing, and (<i>m</i> , <i>ldc</i>) for zero-based indexing. The leading <i>m</i> -by- <i>n</i> part of the array <i>c</i> contains the output matrix <i>C</i> .
----------	---

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, n, ldb, ldc, nnz
  INTEGER      rowind(*), colind(*)
  REAL         alpha
  REAL         val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE PRECISION      alpha
  DOUBLE PRECISION      val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_ccoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc, nnz
  INTEGER       rowind(*), colind(*)
  COMPLEX      alpha
  COMPLEX      val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zcoosm(transa, m, n, alpha, matdescra, val, rowind, colind, nnz, b, ldb, c, ldc)
  CHARACTER*1   transa
  CHARACTER     matdescra(*)
  INTEGER       m, n, ldb, ldc, nnz
  INTEGER       rowind(*), colind(*)
  DOUBLE COMPLEX      alpha
  DOUBLE COMPLEX      val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_scoosm(char *transa, int *m, int *n, float *alpha, char *matdescra,
float *val, int *rowind, int *colind, int *nnz, float *b, int *ldb, float *c, int *ldc);
void mkl_dcoosm(char *transa, int *m, int *n, double *alpha, char *matdescra,
double *val, int *rowind, int *colind, int *nnz, double *b, int *ldb, double *c, int *ldc);
void mkl_ccoosm(char *transa, int *m, int *n, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, int *rowind, int *colind, int *nnz, MKL_Complex8 *b, int *ldb, MKL_Complex8 *c, int
*ldc);
void mkl_zcoosm(char *transa, int *m, int *n, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, int *rowind, int *colind, int *nnz, MKL_Complex16 *b, int *ldb, MKL_Complex16 *c,
int *ldc);
```

mkl_?bsrsm

Solves a system of linear matrix equations for a sparse matrix in the BSR format.

Syntax

Fortran:

```
call mkl_scsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
c, ldc)

call mkl_dcsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
c, ldc)

call mkl_ccsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
c, ldc)

call mkl_zcsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb,
c, ldc)
```

C:

```
void mkl_sbsrsm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *lb, float *alpha, char
*matdescra, float *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, float *b,
MKL_INT *ldb, float *c, MKL_INT *ldc);

void mkl_dbsrsm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *lb, double *alpha,
char *matdescra, double *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT *pntre, double
*b, MKL_INT *ldb, double *c, MKL_INT *ldc);

void mkl_cbsrsm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *lb, MKL_Complex8
*alpha, char *matdescra, MKL_Complex8 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT
*pntre, MKL_Complex8 *b, MKL_INT *ldb, MKL_Complex8 *c, MKL_INT *ldc);

void mkl_zbsrsm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *lb, MKL_Complex16
*alpha, char *matdescra, MKL_Complex16 *val, MKL_INT *indx, MKL_INT *pntrb, MKL_INT
*pntre, MKL_Complex16 *b, MKL_INT *ldb, MKL_Complex16 *c, MKL_INT *ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?bsrsm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the BSR format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(A')*B,
```

where:

`alpha` is scalar, `B` and `C` are dense matrices, `A` is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, `A'` is the transpose of `A`.

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then the matrix-matrix product is computed as $C := \alpha * \text{inv}(A) * B$. If <i>transa</i> = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as $C := \alpha * \text{inv}(A') * B$.
<i>m</i>	INTEGER. Number of block columns of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>lb</i>	INTEGER. Size of the block in the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_sbsrsm. DOUBLE PRECISION for mkl_dbsrsm. COMPLEX for mkl_cbsrsm. DOUBLE COMPLEX for mkl_zbsrsm. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra" .
<i>val</i>	REAL for mkl_sbsrsm. DOUBLE PRECISION for mkl_dbsrsm. COMPLEX for mkl_cbsrsm. DOUBLE COMPLEX for mkl_zbsrsm. Array containing elements of non-zero blocks of the matrix <i>A</i> . Its length is equal to the <i>ABAB</i> number <i>ABAB</i> of non-zero blocks in the matrix <i>A</i> multiplied by <i>lb</i> * <i>lb</i> . Refer to the <i>values</i> array description in BSR Format for more details.
<hr/>	
NOTE	
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).	
No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.	
<hr/>	
<i>indx</i>	INTEGER. Array containing the column indices for each non-zero block in the matrix <i>A</i> . Its length is equal to the number of non-zero blocks in the matrix <i>A</i> . Refer to the <i>columns</i> array description in BSR Format for more details.
<i>pntrb</i>	INTEGER. Array of length <i>m</i> . For one-based indexing: this array contains row indices, such that <i>pntrb</i> (<i>i</i>) - <i>pntrb</i> (1) + 1 is the first index of block row <i>i</i> in the array <i>indx</i> .

For zero-based indexing: this array contains row indices, such that $pntrb(i) - pntrb(0)$ is the first index of block row i in the array $indx$.

Refer to [pointerB](#) array description in [BSR Format](#) for more details.

$pntrc$

INTEGER. Array of length m .

For one-based indexing this array contains row indices, such that $pntrc(i) - pntrb(1)$ is the last index of block row i in the array $indx$.

For zero-based indexing this array contains row indices, such that $pntrc(i) - pntrb(0) - 1$ is the last index of block row i in the array $indx$.

Refer to [pointerE](#) array description in [BSR Format](#) for more details.

b

REAL for `mkl_sbsrsm`.

DOUBLE PRECISION for `mkl_dbsrsm`.

COMPLEX for `mkl_cbsrsm`.

DOUBLE COMPLEX for `mkl_zbsrsm`.

Array, size (ldb, n) for one-based indexing, size (m, ldb) for zero-based indexing.

On entry the leading m -by- n part of the array b must contain the matrix B .

ldb

INTEGER. Specifies the leading dimension (in blocks) of b as declared in the calling (sub)program.

ldc

INTEGER. Specifies the leading dimension (in blocks) of c as declared in the calling (sub)program.

Output Parameters

c

REAL for `mkl_sbsrsm`.

DOUBLE PRECISION for `mkl_dbsrsm`.

COMPLEX for `mkl_cbsrsm`.

DOUBLE COMPLEX for `mkl_zbsrsm`.

Array, size (ldc, n) for one-based indexing, size (m, ldc) for zero-based indexing.

The leading m -by- n part of the array c contains the output matrix C .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntrc, b, ldb, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, lb, ldb, ldc
```

```
INTEGER indx(*), pntrb(m), pntrc(m)
```

```
REAL alpha
```

```
REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
  CHARACTER*1      transa
  CHARACTER        matdescra(*)
  INTEGER          m, n, lb, ldb, ldc
  INTEGER          indx(*), pntrb(m), pntre(m)
  DOUBLE PRECISION alpha
  DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_cbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
  CHARACTER*1      transa
  CHARACTER        matdescra(*)
  INTEGER          m, n, lb, ldb, ldc
  INTEGER          indx(*), pntrb(m), pntre(m)
  COMPLEX          alpha
  COMPLEX          val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zbsrsm(transa, m, n, lb, alpha, matdescra, val, indx, pntrb, pntre, b, ldb, c, ldc)
  CHARACTER*1      transa
  CHARACTER        matdescra(*)
  INTEGER          m, n, lb, ldb, ldc
  INTEGER          indx(*), pntrb(m), pntre(m)
  DOUBLE COMPLEX    alpha
  DOUBLE COMPLEX    val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_sbsrsm(char *transa, int *m, int *n, int *lb, float *alpha, char *matdescra,
float *val, int *indx, int *pntrb, int *pntre, float *b, int *ldb, float *c, int *ldc);
void mkl_dbsrsm(char *transa, int *m, int *n, int *lb, double *alpha, char *matdescra,
double *val, int *indx, int *pntrb, int *pntre, double *b, int *ldb, double *c, int *ldc);
void mkl_cbsrsm(char *transa, int *m, int *n, int *lb, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, int *indx, int *pntrb, int *pntre, MKL_Complex8 *b, int *ldb, MKL_Complex8 *c, int
*ldc);
void mkl_zbsrsm(char *transa, int *m, int *n, int *lb, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, int *indx, int *pntrb, int *pntre, MKL_Complex16 *b, int *ldb, MKL_Complex16 *c,
int *ldc);
```

mkl_?diamv

*Computes matrix - vector product for a sparse matrix
in the diagonal format with one-based indexing.*

Syntax

Fortran:

```
call mkl_sdiamv(transa, m, k, alpha, matdescra, val, lval, iddiag, ndiag, x, beta, y)
call mkl_ddiamv(transa, m, k, alpha, matdescra, val, lval, iddiag, ndiag, x, beta, y)
call mkl_cdiamv(transa, m, k, alpha, matdescra, val, lval, iddiag, ndiag, x, beta, y)
call mkl_zdiamv(transa, m, k, alpha, matdescra, val, lval, iddiag, ndiag, x, beta, y)
```

C:

```
void mkl_sdiamv (char *transa, MKL_INT *m, MKL_INT *k, float *alpha, char *matdescra,
float *val, MKL_INT *lval, MKL_INT *iddiag, MKL_INT *ndiag, float *x, float *beta,
float *y);

void mkl_ddiamv (char *transa, MKL_INT *m, MKL_INT *k, double *alpha, char *matdescra,
double *val, MKL_INT *lval, MKL_INT *iddiag, MKL_INT *ndiag, double *x, double *beta,
double *y);

void mkl_cdiamv (char *transa, MKL_INT *m, MKL_INT *k, MKL_Complex8 *alpha, char
*matdescra, MKL_Complex8 *val, MKL_INT *lval, MKL_INT *iddiag, MKL_INT *ndiag,
MKL_Complex8 *x, MKL_Complex8 *beta, MKL_Complex8 *y);

void mkl_zdiamv (char *transa, MKL_INT *m, MKL_INT *k, MKL_Complex16 *alpha, char
*matdescra, MKL_Complex16 *val, MKL_INT *lval, MKL_INT *iddiag, MKL_INT *ndiag,
MKL_Complex16 *x, MKL_Complex16 *beta, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?diamv` routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A'*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix stored in the diagonal format, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation.
---------------	---------------------------------------

If $\text{transa} = \text{'N'}$ or 'n' , then $y := \alpha * A * x + \beta * y$,
 If $\text{transa} = \text{'T'}$ or 't' or 'C' or 'c' , then $y := \alpha * A' * x + \beta * y$.

m INTEGER. Number of rows of the matrix *A*.

k INTEGER. Number of columns of the matrix *A*.

alpha REAL for mkl_sdiamp.

DOUBLE PRECISION for mkl_ddiamv.

COMPLEX for mkl_cdiamp.

DOUBLE COMPLEX for mkl_zdiamp.

Specifies the scalar *alpha*.

matdescra CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in [Table "Possible Values of the Parameter matdescra \(descra\)"](#). Possible combinations of element values of this parameter are given in [Table "Possible Combinations of Element Values of the Parameter matdescra"](#).

val REAL for mkl_sdiamp.

DOUBLE PRECISION for mkl_ddiamv.

COMPLEX for mkl_cdiamp.

DOUBLE COMPLEX for mkl_zdiamp.

Two-dimensional array of size *lval* by *ndiag*, contains non-zero diagonals of the matrix *A*. Refer to *values* array description in [Diagonal Storage Scheme](#) for more details.

lval INTEGER. Leading dimension of *val*, $lval \geq m$. Refer to *lval* description in [Diagonal Storage Scheme](#) for more details.

idiag INTEGER. Array of length *ndiag*, contains the distances between main diagonal and each non-zero diagonals in the matrix *A*.

Refer to *distance* array description in [Diagonal Storage Scheme](#) for more details.

ndiag INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.

x REAL for mkl_sdiamp.

DOUBLE PRECISION for mkl_ddiamv.

COMPLEX for mkl_cdiamp.

DOUBLE COMPLEX for mkl_zdiamp.

Array, size at least *k* if $\text{transa} = \text{'N'}$ or 'n' , and at least *m* otherwise. On entry, the array *x* must contain the vector *x*.

beta REAL for mkl_sdiamp.

DOUBLE PRECISION for mkl_ddiamv.

COMPLEX for mkl_cdiamv.
 DOUBLE COMPLEX for mkl_zdiamv.
 Specifies the scalar *beta*.

y
 REAL for mkl_sdiamv.
 DOUBLE PRECISION for mkl_ddiamv.
 COMPLEX for mkl_cdiamv.
 DOUBLE COMPLEX for mkl_zdiamv.
 Array, size at least *m* if *transa* = 'N' or 'n', and at least *k* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y Overwritten by the updated vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiamv(transa, m, k, alpha, matdescra, val, lval, iddiag,
ndiag, x, beta, y)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, k, lval, ndiag
  INTEGER iddiag(*)
  REAL alpha, beta
  REAL val(lval,*), x(*), y(*)

SUBROUTINE mkl_ddiamv(transa, m, k, alpha, matdescra, val, lval, iddiag,
ndiag, x, beta, y)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, k, lval, ndiag
  INTEGER iddiag(*)
  DOUBLE PRECISION alpha, beta
  DOUBLE PRECISION val(lval,*), x(*), y(*)
```

```
SUBROUTINE mkl_cdiamv(transa, m, k, alpha, matdescra, val, lval, iddiag,
ndiag, x, beta, y)

  CHARACTER*1   transa

  CHARACTER     matdescra(*)

  INTEGER       m, k, lval, ndiag

  INTEGER       iddiag(*)

  COMPLEX       alpha, beta

  COMPLEX       val(lval,*), x(*), y(*)
```



```
SUBROUTINE mkl_zdiamv(transa, m, k, alpha, matdescra, val, lval, iddiag,
ndiag, x, beta, y)

  CHARACTER*1   transa

  CHARACTER     matdescra(*)

  INTEGER       m, k, lval, ndiag

  INTEGER       iddiag(*)

  DOUBLE COMPLEX       alpha, beta

  DOUBLE COMPLEX       val(lval,*), x(*), y(*)
```

C:

```
void mkl_sdiamv(char *transa, int *m, int *k, float *alpha,
char *matdescra, float *val, int *lval, int *idiag,
int *ndiag, float *x, float *beta, float *y);

void mkl_ddiamv(char *transa, int *m, int *k, double *alpha,
char *matdescra, double *val, int *lval, int *idiag,
int *ndiag, double *x, double *beta, double *y);

void mkl_cdiamv(char *transa, int *m, int *k, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, int *lval, int *idiag,
int *ndiag, MKL_Complex8 *x, MKL_Complex8 *beta, MKL_Complex8 *y);

void mkl_zdiamv(char *transa, int *m, int *k, MKL_Complex16 *alpha,
char *matdescra, MKL_Complex16 *val, int *lval, int *idiag,
int *ndiag, MKL_Complex16 *x, MKL_Complex16 *beta, MKL_Complex16 *y);
```

[mkl_?skymv](#)

Computes matrix - vector product for a sparse matrix in the skyline storage format with one-based indexing.

Syntax**Fortran:**

```
call mkl_sskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

```
call mkl_dskymv(transa, m, k, alpha, matdescra, val, pptr, x, beta, y)
call mkl_cskymv(transa, m, k, alpha, matdescra, val, pptr, x, beta, y)
call mkl_zskymv(transa, m, k, alpha, matdescra, val, pptr, x, beta, y)
```

C:

```
void mkl_sskymv (char *transa, MKL_INT *m, MKL_INT *k, float *alpha, char *matdescra,
float *val, MKL_INT *pptr, float *x, float *beta, float *y);

void mkl_dskymv (char *transa, MKL_INT *m, MKL_INT *k, double *alpha, char *matdescra,
double *val, MKL_INT *pptr, double *x, double *beta, double *y);

void mkl_cskymv (char *transa, MKL_INT *m, MKL_INT *k, MKL_Complex8 *alpha, char
*matdescra, MKL_Complex8 *val, MKL_INT *pptr, MKL_Complex8 *x, MKL_Complex8 *beta,
MKL_Complex8 *y);

void mkl_zskymv (char *transa, MKL_INT *m, MKL_INT *k, MKL_Complex16 *alpha, char
*matdescra, MKL_Complex16 *val, MKL_INT *pptr, MKL_Complex16 *x, MKL_Complex16 *beta,
MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?skymv` routine performs a matrix-vector operation defined as

```
y := alpha*A*x + beta*y
```

or

```
y := alpha*A'*x + beta*y,
```

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *m*-by-*k* sparse matrix stored using the skyline storage scheme, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation.
	If <i>transa</i> = 'N' or 'n', then <i>y</i> := <i>alpha</i> * <i>A</i> * <i>x</i> + <i>beta</i> * <i>y</i>
	If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>y</i> := <i>alpha</i> * <i>A'</i> * <i>x</i> + <i>beta</i> * <i>y</i> ,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .

<i>alpha</i>	REAL for mkl_sskymv. DOUBLE PRECISION for mkl_dskymv. COMPLEX for mkl_cskymv. DOUBLE COMPLEX for mkl_zskymv. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra" .
<hr/>	
	NOTE
	General matrices (<i>matdescra</i> (1) = 'G') is not supported.
<hr/>	
<i>val</i>	REAL for mkl_sskymv. DOUBLE PRECISION for mkl_dskymv. COMPLEX for mkl_cskymv. DOUBLE COMPLEX for mkl_zskymv. Array containing the set of elements of the matrix <i>A</i> in the skyline profile form. If <i>matdescrsa</i> (2) = 'L', then <i>val</i> contains elements from the low triangle of the matrix <i>A</i> . If <i>matdescrsa</i> (2) = 'U', then <i>val</i> contains elements from the upper triangle of the matrix <i>A</i> . Refer to <i>values</i> array description in Skyline Storage Scheme for more details.
<i>pntr</i>	INTEGER. Array of length $(m+m)$ for lower triangle, and $(k+k)$ for upper triangle. It contains the indices specifying in the <i>val</i> the positions of the first element in each row (column) of the matrix <i>A</i> . Refer to <i>pointers</i> array description in Skyline Storage Scheme for more details.
<i>x</i>	REAL for mkl_sskymv. DOUBLE PRECISION for mkl_dskymv. COMPLEX for mkl_cskymv. DOUBLE COMPLEX for mkl_zskymv. Array, size at least <i>k</i> if <i>transa</i> = 'N' or 'n' and at least <i>m</i> otherwise. On entry, the array <i>x</i> must contain the vector <i>x</i> .
<i>beta</i>	REAL for mkl_sskymv. DOUBLE PRECISION for mkl_dskymv.

COMPLEX for mkl_cskymv.
 DOUBLE COMPLEX for mkl_zskymv.
 Specifies the scalar *beta*.

y
 REAL for mkl_sskymv.
 DOUBLE PRECISION for mkl_dskymv.
 COMPLEX for mkl_cskymv.
 DOUBLE COMPLEX for mkl_zskymv.
 Array, size at least *m* if *transa* = 'N' or 'n' and at least *k* otherwise. On entry, the array *y* must contain the vector *y*.

Output Parameters

y Overwritten by the updated vector *y*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m, k
```

```
  INTEGER pntr(*)
```

```
  REAL alpha, beta
```

```
  REAL val(*), x(*), y(*)
```

```
SUBROUTINE mkl_dskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m, k
```

```
  INTEGER pntr(*)
```

```
  DOUBLE PRECISION alpha, beta
```

```
  DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cskymv(transa, m, k, alpha, matdescra, val, pntr, x, beta, y)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m, k
```

```
  INTEGER pntr(*)
```

```
  COMPLEX alpha, beta
```

```
  COMPLEX val(*), x(*), y(*)
```

```
SUBROUTINE mkl_zskymv(transa, m, k, alpha, matdescra, val, pptr, x, beta, y)
CHARACTER*1 transa
CHARACTER matdescra(*)
INTEGER m, k
INTEGER pptr(*)
DOUBLE COMPLEX alpha, beta
DOUBLE COMPLEX val(*), x(*), y(*)
```

C:

```
void mkl_sskymv (char *transa, int *m, int *k, float *alpha, char *matdescra,
float *val, int *pptr, float *x, float *beta, float *y);

void mkl_dskymv (char *transa, int *m, int *k, double *alpha, char *matdescra,
double *val, int *pptr, double *x, double *beta, double *y);

void mkl_cskymv (char *transa, int *m, int *k, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, int *pptr, MKL_Complex8 *x, MKL_Complex8 *beta, MKL_Complex8 *y);

void mkl_zskymv (char *transa, int *m, int *k, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, int *pptr, MKL_Complex16 *x, MKL_Complex16 *beta, MKL_Complex16 *y);
```

mkl_?diasv

Solves a system of linear equations for a sparse matrix in the diagonal format with one-based indexing.

Syntax**Fortran:**

```
call mkl_sdiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
call mkl_ddiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
call mkl_cdiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
call mkl_zdiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
```

C:

```
void mkl_sdiasv (char *transa, MKL_INT *m, float *alpha, char *matdescra, float *val,
MKL_INT *lval, MKL_INT *iddiag, MKL_INT *ndiag, float *x, float *y);
void mkl_ddiasv (char *transa, MKL_INT *m, double *alpha, char *matdescra, double *val,
MKL_INT *lval, MKL_INT *iddiag, MKL_INT *ndiag, double *x, double *y);
void mkl_cdiasv (char *transa, MKL_INT *m, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, MKL_INT *lval, MKL_INT *iddiag, MKL_INT *ndiag, MKL_Complex8 *x,
MKL_Complex8 *y);
void mkl_zdiasv (char *transa, MKL_INT *m, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, MKL_INT *lval, MKL_INT *iddiag, MKL_INT *ndiag, MKL_Complex16 *x,
MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?diasv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A')* x,
```

where:

alpha is scalar, *x* and *y* are vectors, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then <i>y</i> := <i>alpha</i> *inv(<i>A</i>)* <i>x</i> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>y</i> := <i>alpha</i> *inv(<i>A'</i>)* <i>x</i> ,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>alpha</i>	REAL for <code>mkl_sdiasv</code> . DOUBLE PRECISION for <code>mkl_ddiasv</code> . COMPLEX for <code>mkl_cdiasv</code> . DOUBLE COMPLEX for <code>mkl_zdiasv</code> . Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for <code>mkl_sdiasv</code> . DOUBLE PRECISION for <code>mkl_ddiasv</code> . COMPLEX for <code>mkl_cdiasv</code> . DOUBLE COMPLEX for <code>mkl_zdiasv</code> .

Two-dimensional array of size $lval$ by $ndiag$, contains non-zero diagonals of the matrix A. Refer to *values* array description in [Diagonal Storage Scheme](#) for more details.

lval INTEGER. Leading dimension of *val*, $lval \geq m$. Refer to *lval* description in [Diagonal Storage Scheme](#) for more details.

idiag INTEGER. Array of length $ndiag$, contains the distances between main diagonal and each non-zero diagonals in the matrix A.

NOTE

All elements of this array must be sorted in increasing order.

Refer to *distance* array description in [Diagonal Storage Scheme](#) for more details.

ndiag INTEGER. Specifies the number of non-zero diagonals of the matrix A.

x REAL for `mkl_sdiasv`.

DOUBLE PRECISION for `mkl_ddiasv`.

COMPLEX for `mkl_cdiasv`.

DOUBLE COMPLEX for `mkl_zdiasv`.

Array, size at least m .

On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y REAL for `mkl_sdiasv`.

DOUBLE PRECISION for `mkl_ddiasv`.

COMPLEX for `mkl_cdiasv`.

DOUBLE COMPLEX for `mkl_zdiasv`.

Array, size at least m .

On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y Contains solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
  CHARACTER*1    transa
  CHARACTER      matdescra(*)
  INTEGER        m, lval, ndiag
  INTEGER        iddiag(*)
  REAL           alpha
  REAL           val(lval,*), x(*), y(*)

SUBROUTINE mkl_ddiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
  CHARACTER*1    transa
  CHARACTER      matdescra(*)
  INTEGER        m, lval, ndiag
  INTEGER        iddiag(*)
  DOUBLE PRECISION   alpha
  DOUBLE PRECISION   val(lval,*), x(*), y(*)

SUBROUTINE mkl_cdiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
  CHARACTER*1    transa
  CHARACTER      matdescra(*)
  INTEGER        m, lval, ndiag
  INTEGER        iddiag(*)
  COMPLEX         alpha
  COMPLEX         val(lval,*), x(*), y(*)

SUBROUTINE mkl_zdiasv(transa, m, alpha, matdescra, val, lval, iddiag, ndiag, x, y)
  CHARACTER*1    transa
  CHARACTER      matdescra(*)
  INTEGER        m, lval, ndiag
  INTEGER        iddiag(*)
  DOUBLE COMPLEX   alpha
  DOUBLE COMPLEX   val(lval,*), x(*), y(*)
```

C:

```
void mkl_sdiasv(char *transa, int *m, float *alpha, char *matdescra,
float *val, int *lval, int *idiag, int *ndiag, float *x, float *y);
void mkl_ddiasv(char *transa, int *m, double *alpha, char *matdescra,
double *val, int *lval, int *idiag, int *ndiag, double *x, double *y);
```

```
void mkl_cdiasv(char *transa, int *m, MKL_Complex8 *alpha, char *matdescra,
                  MKL_Complex8 *val, int *lval, int *idiag, int *ndiag, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zdiasv(char *transa, int *m, MKL_Complex16 *alpha, char *matdescra,
                  MKL_Complex16 *val, int *lval, int *idiag, int *ndiag, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?skysv

Solves a system of linear equations for a sparse matrix in the skyline format with one-based indexing.

Syntax

Fortran:

```
call mkl_ssksysv(transa, m, alpha, matdescra, val, pptr, x, y)
call mkl_dskysv(transa, m, alpha, matdescra, val, pptr, x, y)
call mkl_cskysv(transa, m, alpha, matdescra, val, pptr, x, y)
call mkl_zskysv(transa, m, alpha, matdescra, val, pptr, x, y)
```

C:

```
void mkl_ssksysv (char *transa, MKL_INT *m, float *alpha, char *matdescra, float *val,
                  MKL_INT *pptr, float *x, float *y);
void mkl_dskysv (char *transa, MKL_INT *m, double *alpha, char *matdescra, double *val,
                  MKL_INT *pptr, double *x, double *y);
void mkl_cskysv (char *transa, MKL_INT *m, MKL_Complex8 *alpha, char *matdescra,
                  MKL_Complex8 *val, MKL_INT *pptr, MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zskysv (char *transa, MKL_INT *m, MKL_Complex16 *alpha, char *matdescra,
                  MKL_Complex16 *val, MKL_INT *pptr, MKL_Complex16 *x, MKL_Complex16 *y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?skysv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the skyline storage format:

```
y := alpha*inv(A)*x
```

or

```
y := alpha*inv(A')*x,
```

where:

`alpha` is scalar, `x` and `y` are vectors, `A` is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, `A'` is the transpose of `A`.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then <i>y</i> := <i>alpha</i> *inv(<i>A</i>)* <i>x</i> If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>y</i> := <i>alpha</i> *inv(<i>A'</i>)* <i>x</i> ,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_sskysv. DOUBLE PRECISION for mkl_dskysv. COMPLEX for mkl_cskysv. DOUBLE COMPLEX for mkl_zskysv. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .

NOTE

General matrices (*matdescra*(1)='G') is not supported.

<i>val</i>	REAL for mkl_sskysv. DOUBLE PRECISION for mkl_dskysv. COMPLEX for mkl_cskysv. DOUBLE COMPLEX for mkl_zskysv. Array containing the set of elements of the matrix <i>A</i> in the skyline profile form. If <i>matdescra</i> (2)= 'L', then <i>val</i> contains elements from the low triangle of the matrix <i>A</i> . If <i>matdescra</i> (2)= 'U', then <i>val</i> contains elements from the upper triangle of the matrix <i>A</i> . Refer to <i>values</i> array description in Skyline Storage Scheme for more details.
<i>pntr</i>	INTEGER. Array of length $(m+m)$ for lower triangle, and $(k+k)$ for upper triangle. It contains the indices specifying in the <i>val</i> the positions of the first element in each row (column) of the matrix <i>A</i> . Refer to <i>pointers</i> array description in Skyline Storage Scheme for more details.

x REAL for mkl_sskysv.
 DOUBLE PRECISION for mkl_dskysv.
 COMPLEX for mkl_cskysv.
 DOUBLE COMPLEX for mkl_zskysv.
 Array, size at least m .
 On entry, the array *x* must contain the vector *x*. The elements are accessed with unit increment.

y REAL for mkl_sskysv.
 DOUBLE PRECISION for mkl_dskysv.
 COMPLEX for mkl_cskysv.
 DOUBLE COMPLEX for mkl_zskysv.
 Array, size at least m .
 On entry, the array *y* must contain the vector *y*. The elements are accessed with unit increment.

Output Parameters

y Contains solution vector *x*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sskysv(transa, m, alpha, matdescra, val, pntr, x, y)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m
  INTEGER      pntr(*)
  REAL         alpha
  REAL         val(*), x(*), y(*)

SUBROUTINE mkl_dskysv(transa, m, alpha, matdescra, val, pntr, x, y)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m
  INTEGER      pntr(*)
  DOUBLE PRECISION alpha
  DOUBLE PRECISION val(*), x(*), y(*)
```

```
SUBROUTINE mkl_cskysv(transa, m, alpha, matdescra, val, pptr, x, y)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m
  INTEGER pptr(*)
  COMPLEX alpha
  COMPLEX val(*), x(*), y(*)

SUBROUTINE mkl_zskysv(transa, m, alpha, matdescra, val, pptr, x, y)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m
  INTEGER pptr(*)
  DOUBLE COMPLEX alpha
  DOUBLE COMPLEX val(*), x(*), y(*)
```

C:

```
void mkl_sskysv(char *transa, int *m, float *alpha, char *matdescra,
float *val, int *pptr, float *x, float *y);

void mkl_dskysv(char *transa, int *m, double *alpha, char *matdescra,
double *val, int *pptr, double *x, double *y);

void mkl_cskysv(char *transa, int *m, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, int *pptr, MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zskysv(char *transa, int *m, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, int *pptr, MKL_Complex16 *x, MKL_Complex16 *y);
```

mkl_?diamm

Computes matrix-matrix product of a sparse matrix stored in the diagonal format with one-based indexing.

Syntax**Fortran:**

```
call mkl_sdiamm(transa, m, n, k, alpha, matdescra, val, lval, iddiag, ndiag, b, ldb,
beta, c, ldc)

call mkl_ddiamm(transa, m, n, k, alpha, matdescra, val, lval, iddiag, ndiag, b, ldb,
beta, c, ldc)

call mkl_cdiamm(transa, m, n, k, alpha, matdescra, val, lval, iddiag, ndiag, b, ldb,
beta, c, ldc)

call mkl_zdiamm(transa, m, n, k, alpha, matdescra, val, lval, iddiag, ndiag, b, ldb,
beta, c, ldc)
```

C:

```
void mkl_sdiamm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *alpha, char
*matdescra, float *val, MKL_INT *lval, MKL_INT *idiag, MKL_INT *ndiag, float *b,
MKL_INT *ldb, float *beta, float *c, MKL_INT *ldc);

void mkl_ddiamm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *alpha, char
*matdescra, double *val, MKL_INT *lval, MKL_INT *idiag, MKL_INT *ndiag, double *b,
MKL_INT *ldb, double *beta, double *c, MKL_INT *ldc);

void mkl_cdiamm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, MKL_INT *lval, MKL_INT *idiag, MKL_INT *ndiag,
MKL_Complex8 *b, MKL_INT *ldb, MKL_Complex8 *beta, MKL_Complex8 *c, MKL_INT *ldc);

void mkl_zdiamm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16
*alpha, char *matdescra, MKL_Complex16 *val, MKL_INT *lval, MKL_INT *idiag, MKL_INT
*ndiag, MKL_Complex16 *b, MKL_INT *ldb, MKL_Complex16 *beta, MKL_Complex16 *c, MKL_INT
*ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?diamm` routine performs a matrix-matrix operation defined as

$$C := \alpha A^* B + \beta C$$

or

$$C := \alpha A'^* B + \beta C,$$

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the diagonal format, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation. If <i>transa</i> = 'N' or 'n', then $C := \alpha A^* B + \beta C$, If <i>transa</i> = 'T' or 't' or 'C' or 'c', then $C := \alpha A'^* B + \beta C$.
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .

<i>alpha</i>	REAL for mkl_sdiamm. DOUBLE PRECISION for mkl_ddiamm. COMPLEX for mkl_cdiamm. DOUBLE COMPLEX for mkl_zdiamm. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for mkl_sdiamm. DOUBLE PRECISION for mkl_ddiamm. COMPLEX for mkl_cdiamm. DOUBLE COMPLEX for mkl_zdiamm. Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , <i>lval</i> \geq <i>m</i> . Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> . Refer to <i>distance</i> array description in Diagonal Storage Scheme for more details.
<i>ndiag</i>	INTEGER. Specifies the number of non-zero diagonals of the matrix <i>A</i> .
<i>b</i>	REAL for mkl_sdiamm. DOUBLE PRECISION for mkl_ddiamm. COMPLEX for mkl_cdiamm. DOUBLE COMPLEX for mkl_zdiamm. Array, size (<i>ldb</i> , <i>n</i>). On entry with <i>transa</i> = 'N' or 'n', the leading <i>k</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> , otherwise the leading <i>m</i> -by- <i>n</i> part of the array <i>b</i> must contain the matrix <i>B</i> .
<i>ldb</i>	INTEGER. Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program.
<i>beta</i>	REAL for mkl_sdiamm. DOUBLE PRECISION for mkl_ddiamm. COMPLEX for mkl_cdiamm. DOUBLE COMPLEX for mkl_zdiamm.

Specifies the scalar *beta*.

c REAL for mkl_sdiamm.

DOUBLE PRECISION for mkl_ddiamm.

COMPLEX for mkl_cdiamm.

DOUBLE COMPLEX for mkl_zdiamm.

Array, size *ldc* by *n*.

On entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

Output Parameters

c Overwritten by the matrix (*alpha***A***B* + *beta***C*) or (*alpha***A*'**B* + *beta***C*).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiamm(transa, m, n, k, alpha, matdescra, val, lval,
idiag, ndiag, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc, lval, ndiag
```

```
INTEGER idiag(*)
```

```
REAL alpha, beta
```

```
REAL val(lval,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_ddiamm(transa, m, n, k, alpha, matdescra, val, lval,
idiag, ndiag, b, ldb, beta, c, ldc)
```

```
CHARACTER*1 transa
```

```
CHARACTER matdescra(*)
```

```
INTEGER m, n, k, ldb, ldc, lval, ndiag
```

```
INTEGER idiag(*)
```

```
DOUBLE PRECISION alpha, beta
```

```
DOUBLE PRECISION val(lval,*), b(ldb,*), c(ldc,*)
```

```

SUBROUTINE mkl_cdiamm(transa, m, n, k, alpha, matdescra, val, lval,
  iddiag, ndiag, b, ldb, beta, c, ldc)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, n, k, ldb, ldc, lval, ndiag
  INTEGER iddiag(*)
  COMPLEX alpha, beta
  COMPLEX val(lval,*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zdiamm(transa, m, n, k, alpha, matdescra, val, lval,
  iddiag, ndiag, b, ldb, beta, c, ldc)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, n, k, ldb, ldc, lval, ndiag
  INTEGER iddiag(*)
  DOUBLE COMPLEX alpha, beta
  DOUBLE COMPLEX val(lval,*), b(ldb,*), c(ldc,*)

```

C:

```

void mkl_sdiamm(char *transa, int *m, int *n, int *k, float *alpha,
  char *matdescra, float *val, int *lval, int *idiag, int *ndiag,
  float *b, int *ldb, float *beta, float *c, int *ldc);

void mkl_ddiamm(char *transa, int *m, int *n, int *k, double *alpha,
  char *matdescra, double *val, int *lval, int *idiag, int *ndiag,
  double *b, int *ldb, double *beta, double *c, int *ldc);

void mkl_cdiamm(char *transa, int *m, int *n, int *k, MKL_Complex8 *alpha,
  char *matdescra, MKL_Complex8 *val, int *lval, int *idiag, int *ndiag,
  MKL_Complex8 *b, int *ldb, MKL_Complex8 *beta, MKL_Complex8 *c, int *ldc);

void mkl_zdiamm(char *transa, int *m, int *n, int *k, MKL_Complex16 *alpha,
  char *matdescra, MKL_Complex16 *val, int *lval, int *idiag, int *ndiag,
  MKL_Complex16 *b, int *ldb, MKL_Complex16 *beta, MKL_Complex16 *c, int *ldc);

```

mkl_?skymm

Computes matrix-matrix product of a sparse matrix stored using the skyline storage scheme with one-based indexing.

Syntax**Fortran:**

```
call mkl_sskymm(transa, m, n, k, alpha, matdescra, val, pptr, b, ldb, beta, c, ldc)
call mkl_dskymm(transa, m, n, k, alpha, matdescra, val, pptr, b, ldb, beta, c, ldc)
call mkl_cskymm(transa, m, n, k, alpha, matdescra, val, pptr, b, ldb, beta, c, ldc)
call mkl_zskymm(transa, m, n, k, alpha, matdescra, val, pptr, b, ldb, beta, c, ldc)
```

C:

```
void mkl_sskymm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *alpha, char
*matdescra, float *val, MKL_INT *pptr, float *b, MKL_INT *ldb, float *beta, float *c,
MKL_INT *ldc);

void mkl_dskymm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *alpha, char
*matdescra, double *val, MKL_INT *pptr, double *b, MKL_INT *ldb, double *beta, double
*c, MKL_INT *ldc);

void mkl_cskymm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, MKL_INT *pptr, MKL_Complex8 *b, MKL_INT *ldb,
MKL_Complex8 *beta, MKL_Complex8 *c, MKL_INT *ldc);

void mkl_zskymm (char *transa, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16
*alpha, char *matdescra, MKL_Complex16 *val, MKL_INT *pptr, MKL_Complex16 *b, MKL_INT
*ldb, MKL_Complex16 *beta, MKL_Complex16 *c, MKL_INT *ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?skymm` routine performs a matrix-matrix operation defined as

```
C := alpha*A*B + beta*C
```

or

```
C := alpha*A'*B + beta*C,
```

where:

alpha and *beta* are scalars,

B and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the skyline storage format, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the operation.
	If <i>transa</i> = 'N' or 'n', then <i>C</i> := <i>alpha</i> * <i>A</i> * <i>B</i> + <i>beta</i> * <i>C</i> ,
	If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>C</i> := <i>alpha</i> * <i>A'</i> * <i>B</i> + <i>beta</i> * <i>C</i> ,

<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>alpha</i>	REAL for mkl_sskymm. DOUBLE PRECISION for mkl_dskymm. COMPLEX for mkl_cskymm. DOUBLE COMPLEX for mkl_zskymm. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .

NOTE

General matrices (*matdescra* (1)='G') is not supported.

<i>val</i>	REAL for mkl_sskymm. DOUBLE PRECISION for mkl_dskymm. COMPLEX for mkl_cskymm. DOUBLE COMPLEX for mkl_zskymm. Array containing the set of elements of the matrix <i>A</i> in the skyline profile form. If <i>matdescrsa</i> (2)= 'L', then <i>val</i> contains elements from the low triangle of the matrix <i>A</i> . If <i>matdescrsa</i> (2)= 'U', then <i>val</i> contains elements from the upper triangle of the matrix <i>A</i> . Refer to <i>values</i> array description in Skyline Storage Scheme for more details.
<i>pntr</i>	INTEGER. Array of length $(m+m)$ for lower triangle, and $(k+k)$ for upper triangle. It contains the indices specifying in the <i>val</i> the positions of the first element in each row (column) of the matrix <i>A</i> . Refer to <i>pointers</i> array description in Skyline Storage Scheme for more details.
<i>b</i>	REAL for mkl_sskymm. DOUBLE PRECISION for mkl_dskymm. COMPLEX for mkl_cskymm. DOUBLE COMPLEX for mkl_zskymm.

Array, size (*ldb*, *n*).

On entry with *transa* = 'N' or 'n', the leading *k*-by-*n* part of the array *b* must contain the matrix *B*, otherwise the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

beta REAL for mkl_ssksymm.

DOUBLE PRECISION for mkl_dskymm.

COMPLEX for mkl_cskymm.

DOUBLE COMPLEX for mkl_zskymm.

Specifies the scalar *beta*.

c REAL for mkl_ssksymm.

DOUBLE PRECISION for mkl_dskymm.

COMPLEX for mkl_cskymm.

DOUBLE COMPLEX for mkl_zskymm.

Array, size *ldc* by *n*.

On entry, the leading *m*-by-*n* part of the array *c* must contain the matrix *C*, otherwise the leading *k*-by-*n* part of the array *c* must contain the matrix *C*.

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

Output Parameters

c Overwritten by the matrix (*alpha***A***B* + *beta***C*) or (*alpha***A*'**B* + *beta***C*).

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_ssksymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
  ldb, beta, c, ldc)
  CHARACTER*1  transa
  CHARACTER    matdescra(*)
  INTEGER      m, n, k, ldb, ldc
  INTEGER      pntr(*)
  REAL         alpha, beta
  REAL         val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
    ldb, beta, c, ldc)
```

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, k, ldb, ldc

INTEGER pntr(*)

DOUBLE PRECISION alpha, beta

DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)

```
SUBROUTINE mkl_cskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
    ldb, beta, c, ldc)
```

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, k, ldb, ldc

INTEGER pntr(*)

COMPLEX alpha, beta

COMPLEX val(*), b(ldb,*), c(ldc,*)

```
SUBROUTINE mkl_zskymm(transa, m, n, k, alpha, matdescra, val, pntr, b,
    ldb, beta, c, ldc)
```

CHARACTER*1 transa

CHARACTER matdescra(*)

INTEGER m, n, k, ldb, ldc

INTEGER pntr(*)

DOUBLE COMPLEX alpha, beta

DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)

C:

```
void mkl_sskymm(char *transa, int *m, int *n, int *k, float *alpha,
char *matdescra, float *val, int *pntr, float *b, int *ldb,
float *beta, float *c, int *ldc);

void mkl_dskymm(char *transa, int *m, int *n, int *k, double *alpha,
char *matdescra, double *val, int *pntr, double *b, int *ldb,
double *beta, double *c, int *ldc);

void mkl_cskymm(char *transa, int *m, int *n, int *k, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, int *pntr, MKL_Complex8 *b, int *ldb,
MKL_Complex8 *beta, MKL_Complex8 *c, int *ldc);
```

```
void mkl_zskymm(char *transa, int *m, int *n, int *k, MKL_Complex16 *alpha,
char *matdescra, MKL_Complex16 *val, int *pntr, MKL_Complex16 *b, int *ldb,
MKL_Complex16 *beta, MKL_Complex16 *c, int *ldc);
```

mkl_?diasm

Solves a system of linear matrix equations for a sparse matrix in the diagonal format with one-based indexing.

Syntax

Fortran:

```
call mkl_sdiasm(transa, m, n, alpha, matdescra, val, lval, iddiag, ndiag, b, ldb, c,
ldc)

call mkl_ddiasm(transa, m, n, alpha, matdescra, val, lval, iddiag, ndiag, b, ldb, c,
ldc)

call mkl_cdiasm(transa, m, n, alpha, matdescra, val, lval, iddiag, ndiag, b, ldb, c,
ldc)

call mkl_zdiasm(transa, m, n, alpha, matdescra, val, lval, iddiag, ndiag, b, ldb, c,
ldc)
```

C:

```
void mkl_sdiasm (char *transa, MKL_INT *m, MKL_INT *n, float *alpha, char *matdescra,
float *val, MKL_INT *lval, MKL_INT *idiag, MKL_INT *ndiag, float *b, MKL_INT *ldb,
float *c, MKL_INT *ldc);

void mkl_ddiasm (char *transa, MKL_INT *m, MKL_INT *n, double *alpha, char *matdescra,
double *val, MKL_INT *lval, MKL_INT *idiag, MKL_INT *ndiag, double *b, MKL_INT *ldb,
double *c, MKL_INT *ldc);

void mkl_cdiasm (char *transa, MKL_INT *m, MKL_INT *n, MKL_Complex8 *alpha, char
*matdescra, MKL_Complex8 *val, MKL_INT *lval, MKL_INT *idiag, MKL_INT *ndiag,
MKL_Complex8 *b, MKL_INT *ldb, MKL_Complex8 *c, MKL_INT *ldc);

void mkl_zdiasm (char *transa, MKL_INT *m, MKL_INT *n, MKL_Complex16 *alpha, char
*matdescra, MKL_Complex16 *val, MKL_INT *lval, MKL_INT *idiag, MKL_INT *ndiag,
MKL_Complex16 *b, MKL_INT *ldb, MKL_Complex16 *c, MKL_INT *ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?diasm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the diagonal format:

$C := \alpha \cdot \text{inv}(A) \cdot B$

or

$C := \alpha \cdot \text{inv}(A') \cdot B,$

where:

alpha is scalar, *B* and *C* are dense matrices, *A* is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, *A'* is the transpose of *A*.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then <i>C</i> := <i>alpha</i> *inv(<i>A</i>)* <i>B</i> , If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>C</i> := <i>alpha</i> *inv(<i>A'</i>)* <i>B</i> .
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	REAL for mkl_sdiasm. DOUBLE PRECISION for mkl_ddiasm. COMPLEX for mkl_cdiasm. DOUBLE COMPLEX for mkl_zdiasm. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .
<i>val</i>	REAL for mkl_sdiasm. DOUBLE PRECISION for mkl_ddiasm. COMPLEX for mkl_cdiasm. DOUBLE COMPLEX for mkl_zdiasm. Two-dimensional array of size <i>lval</i> by <i>ndiag</i> , contains non-zero diagonals of the matrix <i>A</i> . Refer to <i>values</i> array description in Diagonal Storage Scheme for more details.
<i>lval</i>	INTEGER. Leading dimension of <i>val</i> , <i>lval</i> $\geq m$. Refer to <i>lval</i> description in Diagonal Storage Scheme for more details.
<i>idiag</i>	INTEGER. Array of length <i>ndiag</i> , contains the distances between main diagonal and each non-zero diagonals in the matrix <i>A</i> .

NOTE

All elements of this array must be sorted in increasing order.

Refer to *distance* array description in [Diagonal Storage Scheme](#) for more details.

ndiag INTEGER. Specifies the number of non-zero diagonals of the matrix *A*.

b REAL for mkl_sdiasm.

DOUBLE PRECISION for mkl_ddiasm.

COMPLEX for mkl_cdiasm.

DOUBLE COMPLEX for mkl_zdiasm.

Array, size (*ldb*, *n*).

On entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

ldc INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

Output Parameters

c REAL for mkl_sdiasm.

DOUBLE PRECISION for mkl_ddiasm.

COMPLEX for mkl_cdiasm.

DOUBLE COMPLEX for mkl_zdiasm.

Array, size *ldc* by *n*.

The leading *m*-by-*n* part of the array *c* contains the matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdiasm(transa, m, n, alpha, matdescra, val, lval, iddiag,
ndiag, b, ldb, c, ldc)
CHARACTER*1 transa
CHARACTER matdescra(*)
INTEGER m, n, ldb, ldc, lval, ndiag
INTEGER iddiag(*)
REAL alpha
REAL val(lval,*), b(ldb,*), c(ldc,*)
```

```

SUBROUTINE mkl_ddiasm(transa, m, n, alpha, matdescra, val, lval, iddiag,
ndiag, b, ldb, c, ldc)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, n, ldb, ldc, lval, ndiag
  INTEGER iddiag(*)
  DOUBLE PRECISION alpha
  DOUBLE PRECISION val(lval,*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_cdiasm(transa, m, n, alpha, matdescra, val, lval, iddiag,
ndiag, b, ldb, c, ldc)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, n, ldb, ldc, lval, ndiag
  INTEGER iddiag(*)
  COMPLEX alpha
  COMPLEX val(lval,*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zdiasm(transa, m, n, alpha, matdescra, val, lval, iddiag,
ndiag, b, ldb, c, ldc)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, n, ldb, ldc, lval, ndiag
  INTEGER iddiag(*)
  DOUBLE COMPLEX alpha
  DOUBLE COMPLEX val(lval,*), b(ldb,*), c(ldc,*)

```

C:

```

void mkl_sdiasm(char *transa, int *m, int *n, float *alpha,
char *matdescra, float *val, int *lval, int *idiag, int *ndiag,
float *b, int *ldb, float *c, int *ldc);

void mkl_ddiasm(char *transa, int *m, int *n, double *alpha,
char *matdescra, double *val, int *lval, int *idiag, int *ndiag,
double *b, int *ldb, double *c, int *ldc);

void mkl_cdiasm(char *transa, int *m, int *n, MKL_Complex8 *alpha,
char *matdescra, MKL_Complex8 *val, int *lval, int *idiag, int *ndiag,
MKL_Complex8 *b, int *ldb, MKL_Complex8 *c, int *ldc);

```

```
void mkl_zdiasm(char *transa, int *m, int *n, MKL_Complex16 *alpha,
char *matdescra, MKL_Complex16 *val, int *lval, int *idiag, int *ndiag,
MKL_Complex16 *b, int *ldb, MKL_Complex16 *c, int *ldc);
```

mkl_?skysm

Solves a system of linear matrix equations for a sparse matrix stored using the skyline storage scheme with one-based indexing.

Syntax

Fortran:

```
call mkl_sskysm(transa, m, n, alpha, matdescra, val, pptr, b, ldb, c, ldc)
call mkl_dskysm(transa, m, n, alpha, matdescra, val, pptr, b, ldb, c, ldc)
call mkl_cskysm(transa, m, n, alpha, matdescra, val, pptr, b, ldb, c, ldc)
call mkl_zskysm(transa, m, n, alpha, matdescra, val, pptr, b, ldb, c, ldc)
```

C:

```
void mkl_sskysm (char *transa, MKL_INT *m, MKL_INT *n, float *alpha, char *matdescra,
float *val, MKL_INT *pptr, float *b, MKL_INT *ldb, float *c, MKL_INT *ldc);
void mkl_dskysm (char *transa, MKL_INT *m, MKL_INT *n, double *alpha, char *matdescra,
double *val, MKL_INT *pptr, double *b, MKL_INT *ldb, double *c, MKL_INT *ldc);
void mkl_cskysm (char *transa, MKL_INT *m, MKL_INT *n, MKL_Complex8 *alpha, char
*matdescra, MKL_Complex8 *val, MKL_INT *pptr, MKL_Complex8 *b, MKL_INT *ldb,
MKL_Complex8 *c, MKL_INT *ldc);
void mkl_zskysm (char *transa, MKL_INT *m, MKL_INT *n, MKL_Complex16 *alpha, char
*matdescra, MKL_Complex16 *val, MKL_INT *pptr, MKL_Complex16 *b, MKL_INT *ldb,
MKL_Complex16 *c, MKL_INT *ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?skysm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the skyline storage format:

```
C := alpha*inv(A)*B
```

or

```
C := alpha*inv(A')*B,
```

where:

`alpha` is scalar, `B` and `C` are dense matrices, `A` is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, `A'` is the transpose of `A`.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>transa</i>	CHARACTER*1. Specifies the system of linear equations. If <i>transa</i> = 'N' or 'n', then <i>C</i> := <i>alpha</i> *inv(<i>A</i>)* <i>B</i> , If <i>transa</i> = 'T' or 't' or 'C' or 'c', then <i>C</i> := <i>alpha</i> *inv(<i>A'</i>)* <i>B</i> ,
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>C</i> .
<i>alpha</i>	REAL for mkl_sskysm. DOUBLE PRECISION for mkl_dskysm. COMPLEX for mkl_cskysm. DOUBLE COMPLEX for mkl_zskysm. Specifies the scalar <i>alpha</i> .
<i>matdescra</i>	CHARACTER. Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter <i>matdescra</i> (<i>descra</i>)" . Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter <i>matdescra</i>" .

NOTE

General matrices (*matdescra*(1)='G') is not supported.

<i>val</i>	REAL for mkl_sskysm. DOUBLE PRECISION for mkl_dskysm. COMPLEX for mkl_cskysm. DOUBLE COMPLEX for mkl_zskysm. Array containing the set of elements of the matrix <i>A</i> in the skyline profile form. If <i>matdescrsa</i> (2) = 'L', then <i>val</i> contains elements from the low triangle of the matrix <i>A</i> . If <i>matdescrsa</i> (2) = 'U', then <i>val</i> contains elements from the upper triangle of the matrix <i>A</i> . Refer to <i>values</i> array description in Skyline Storage Scheme for more details.
<i>pntr</i>	INTEGER. Array of length $(m+m)$. It contains the indices specifying in the <i>val</i> the positions of the first non-zero element of each <i>i</i> -row (column) of the matrix <i>A</i> such that <i>pointers</i> (<i>i</i>) - <i>pointers</i> (1) + 1. Refer to <i>pointers</i> array description in Skyline Storage Scheme for more details.

b

REAL for mkl_sskysm.
 DOUBLE PRECISION for mkl_dskysm.
 COMPLEX for mkl_cskysm.
 DOUBLE COMPLEX for mkl_zskysm.
 Array, size (*ldb*, *n*).

On entry the leading *m*-by-*n* part of the array *b* must contain the matrix *B*.

ldb

INTEGER. Specifies the leading dimension of *b* as declared in the calling (sub)program.

ldc

INTEGER. Specifies the leading dimension of *c* as declared in the calling (sub)program.

Output Parameters

c

REAL for mkl_sskysm.
 DOUBLE PRECISION for mkl_dskysm.
 COMPLEX for mkl_cskysm.
 DOUBLE COMPLEX for mkl_zskysm.
 Array, size *ldc* by *n*.

The leading *m*-by-*n* part of the array *c* contains the matrix *C*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m, n, ldb, ldc
```

```
  INTEGER pntr(*)
```

```
  REAL alpha
```

```
  REAL val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_dskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
```

```
  CHARACTER*1 transa
```

```
  CHARACTER matdescra(*)
```

```
  INTEGER m, n, ldb, ldc
```

```
  INTEGER pntr(*)
```

```
  DOUBLE PRECISION alpha
```

```
  DOUBLE PRECISION val(*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_cskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, n, ldb, ldc
  INTEGER pntr(*)
  COMPLEX alpha
  COMPLEX val(*), b(ldb,*), c(ldc,*)

SUBROUTINE mkl_zskysm(transa, m, n, alpha, matdescra, val, pntr, b, ldb, c, ldc)
  CHARACTER*1 transa
  CHARACTER matdescra(*)
  INTEGER m, n, ldb, ldc
  INTEGER pntr(*)
  DOUBLE COMPLEX alpha
  DOUBLE COMPLEX val(*), b(ldb,*), c(ldc,*)
```

C:

```
void mkl_sskysm(char *transa, int *m, int *n, float *alpha, char *matdescra,
float *val, int *pntr, float *b, int *ldb, float *c, int *ldc);

void mkl_dskysm(char *transa, int *m, int *n, double *alpha, char *matdescra,
double *val, int *pntr, double *b, int *ldb, double *c, int *ldc);

void mkl_cskysm(char *transa, int *m, int *n, MKL_Complex8 *alpha, char *matdescra,
MKL_Complex8 *val, int *pntr, MKL_Complex8 *b, int *ldb, MKL_Complex8 *c, int *ldc);

void mkl_zskysm(char *transa, int *m, int *n, MKL_Complex16 *alpha, char *matdescra,
MKL_Complex16 *val, int *pntr, MKL_Complex16 *b, int *ldb, MKL_Complex16 *c, int *ldc);
```

mkl_?dnsCSR

Convert a sparse matrix in uncompressed representation to the CSR format and vice versa.

Syntax**Fortran:**

```
call mkl_sdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
call mkl_ddnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
call mkl_cdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
call mkl_zdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
```

C:

```
void mkl_sdnscsr (MKL_INT *job, MKL_INT *m, MKL_INT *n, float *adns, MKL_INT *lda,
float *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *info);
```

```
void mkl_ddnscsr (MKL_INT *job, MKL_INT *m, MKL_INT *n, double *adns, MKL_INT *lda,
double *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *info);

void mkl_cdnscsr (MKL_INT *job, MKL_INT *m, MKL_INT *n, MKL_Complex8 *adns, MKL_INT
*lda, MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *info);

void mkl_zdnscsr (MKL_INT *job, MKL_INT *m, MKL_INT *n, MKL_Complex16 *adns, MKL_INT
*lda, MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *info);
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

This routine converts a sparse matrix A between formats: stored as a rectangular array (dense representation) and stored using compressed sparse row (CSR) format (3-array variation).

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>job</i>	INTEGER
Array, contains the following conversion parameters:	
<i>job(1)</i>	If $job(1)=0$, the rectangular matrix A is converted to the CSR format; if $job(1)=1$, the rectangular matrix A is restored from the CSR format.
<i>job(2)</i>	If $job(2)=0$, zero-based indexing for the rectangular matrix A is used; if $job(2)=1$, one-based indexing for the rectangular matrix A is used.
<i>job(3)</i>	If $job(3)=0$, zero-based indexing for the matrix in CSR format is used; if $job(3)=1$, one-based indexing for the matrix in CSR format is used.
<i>job(4)</i>	If $job(4)=0$, $adns$ is a lower triangular part of matrix A ; If $job(4)=1$, $adns$ is an upper triangular part of matrix A ; If $job(4)=2$, $adns$ is a whole matrix A .
<i>job(5)</i>	$job(5)=nzmax$ - maximum number of the non-zero elements allowed if $job(1)=0$.
<i>job(6)</i>	- job indicator for conversion to CSR format. If $job(6)=0$, only array ia is generated for the output storage. If $job(6)>0$, arrays $acsr$, ia , ja are generated for the output storage.
<i>m</i>	INTEGER. Number of rows of the matrix A .

<i>n</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>adns</i>	(input/output) REAL for mkl_sdnscsr. DOUBLE PRECISION for mkl_ddnscsr. COMPLEX for mkl_cdnscsr. DOUBLE COMPLEX for mkl_zdnscsr. If <i>job</i> (1)=0, on input <i>adns</i> contains an uncompressed (dense) representation of matrix <i>A</i> .
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>adns</i> as declared in the calling (sub)program. For <i>job</i> (2)=0 (zero-based indexing of <i>A</i>), <i>lda</i> must be at least $\max(1, n)$. For <i>job</i> (2)=1 (one-based indexing of <i>A</i>), <i>lda</i> must be at least $\max(1, m)$.
<i>acsr</i>	(input/output) REAL for mkl_sdnscsr. DOUBLE PRECISION for mkl_ddnscsr. COMPLEX for mkl_cdnscsr. DOUBLE COMPLEX for mkl_zdnscsr. If <i>job</i> (1)=1, on input <i>acsr</i> contains the non-zero elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	(input/output) INTEGER. If <i>job</i> (1)=1, on input <i>ja</i> contains the column indices for each non-zero element of the matrix <i>A</i> . Its length is equal to the length of the array <i>acsr</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	(input/output) INTEGER. Array of length $m + 1$. If <i>job</i> (1)=1, on input <i>ia</i> contains indices of elements in the array <i>acsr</i> , such that <i>ia</i> (<i>I</i>) is the index in the array <i>acsr</i> of the first non-zero element from the row <i>I</i> . The value of the last element <i>ia</i> ($m + 1$) is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.

Output Parameters

<i>adns</i>	If <i>job</i> (1)=1, on output <i>adns</i> contains the uncompressed (dense) representation of matrix <i>A</i> .
<i>acsr, ja, ia</i>	If <i>job</i> (1)=0, on output <i>acsr</i> , <i>ja</i> , and <i>ia</i> contain the compressed sparse row (CSR) format (3-array variation) of matrix <i>A</i> (see Sparse Matrix Storage Formats for a description of the storage format).
<i>info</i>	INTEGER. Integer info indicator only for restoring the matrix <i>A</i> from the CSR format.

If $info=0$, the execution is successful.

If $info=i$, the routine is interrupted processing the i -th row because there is no space in the arrays $acsr$ and ja according to the value $nzmax$.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
```

```
    INTEGER      job(8)
```

```
    INTEGER      m, n, lda, info
```

```
    INTEGER      ja(*), ia(m+1)
```

```
    REAL         adns(*), acsr(*)
```

```
SUBROUTINE mkl_ddnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
```

```
    INTEGER      job(8)
```

```
    INTEGER      m, n, lda, info
```

```
    INTEGER      ja(*), ia(m+1)
```

```
    DOUBLE PRECISION      adns(*), acsr(*)
```

```
SUBROUTINE mkl_cdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
```

```
    INTEGER      job(8)
```

```
    INTEGER      m, n, lda, info
```

```
    INTEGER      ja(*), ia(m+1)
```

```
    COMPLEX      adns(*), acsr(*)
```

```
SUBROUTINE mkl_zdnscsr(job, m, n, adns, lda, acsr, ja, ia, info)
```

```
    INTEGER      job(8)
```

```
    INTEGER      m, n, lda, info
```

```
    INTEGER      ja(*), ia(m+1)
```

```
    DOUBLE COMPLEX      adns(*), acsr(*)
```

C:

```
void mkl_sdnscsr(int *job, int *m, int *n, float *adns,
```

```
int *lda, float *acsr, int *ja, int *ia, int *info);
```

```
void mkl_ddnscsr(int *job, int *m, int *n, double *adns,
```

```
int *lda, double *acsr, int *ja, int *ia, int *info);
```

```
void mkl_cdnscsr(int *job, int *m, int *n, MKL_Complex8 *adns,
```

```
int *lda, MKL_Complex8 *acsr, int *ja, int *ia, int *info);
```

```
void mkl_zdnscsr(int *job, int *m, int *n, MKL_Complex16 *adns,
```

```
int *lda, MKL_Complex16 *acsr, int *ja, int *ia, int *info);
```

mkl_?csrcoo

Converts a sparse matrix in the CSR format to the coordinate format and vice versa.

Syntax

Fortran:

```
call mkl_scsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
call mkl_dcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
call mkl_ccsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
call mkl_zcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
```

C:

```
void mkl_dcsrcoo (MKL_INT *job, MKL_INT *n, double *acsr, MKL_INT *ja, MKL_INT *ia,
MKL_INT *nnz, double *aco, MKL_INT *rowind, MKL_INT *colind, MKL_INT *info);
void mkl_scsrcoo (MKL_INT *job, MKL_INT *n, float *acsr, MKL_INT *ja, MKL_INT *ia,
MKL_INT *nnz, float *aco, MKL_INT *rowind, MKL_INT *colind, MKL_INT *info);
void mkl_ccsrcoo (MKL_INT *job, MKL_INT *n, MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT
*ia, MKL_INT *nnz, MKL_Complex8 *aco, MKL_INT *rowind, MKL_INT *colind, MKL_INT
*info);
void mkl_zcsrcoo (MKL_INT *job, MKL_INT *n, MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT
*ia, MKL_INT *nnz, MKL_Complex16 *aco, MKL_INT *rowind, MKL_INT *colind, MKL_INT
*info);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This routine converts a sparse matrix A stored in the compressed sparse row (CSR) format (3-array variation) to coordinate format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>job</i>	INTEGER
	Array, contains the following conversion parameters:
<i>job</i> (1)	If <i>job</i> (1)=0, the matrix in the CSR format is converted to the coordinate format;
	if <i>job</i> (1)=1, the matrix in the coordinate format is converted to the CSR format.
	if <i>job</i> (1)=2, the matrix in the coordinate format is converted to the CSR format, and the column indices in CSR representation are sorted in the increasing order within each row.

job(2)
 If *job*(2)=0, zero-based indexing for the matrix in CSR format is used;
 if *job*(2)=1, one-based indexing for the matrix in CSR format is used.

job(3)
 If *job*(3)=0, zero-based indexing for the matrix in coordinate format is used;
 if *job*(3)=1, one-based indexing for the matrix in coordinate format is used.

job(5)
job(5)=*nzmax* - maximum number of the non-zero elements allowed if *job*(1)=0.
job(5)=*nnz* - sets number of the non-zero elements of the matrix *A* if *job*(1)=1.

job(6) - job indicator.

For conversion to the coordinate format:

If *job*(6)=1, only array *rowind* is filled in for the output storage.
 If *job*(6)=2, arrays *rowind*, *colind* are filled in for the output storage.
 If *job*(6)=3, all arrays *rowind*, *colind*, *acoo* are filled in for the output storage.

For conversion to the CSR format:

If *job*(6)=0, all arrays *acsr*, *ja*, *ia* are filled in for the output storage.
 If *job*(6)=1, only array *ia* is filled in for the output storage.
 If *job*(6)=2, then it is assumed that the routine already has been called with the *job*(6)=1, and the user allocated the required space for storing the output arrays *acsr* and *ja*.

n INTEGER. Dimension of the matrix *A*.

acsr (input/output)

REAL for mkl_scsrcoo.
 DOUBLE PRECISION for mkl_dcsrcoo.
 COMPLEX for mkl_ccsrcoo.
 DOUBLE COMPLEX for mkl_zcsrcoo.

Array containing non-zero elements of the matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja (input/output) INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.

Its length is equal to the length of the array *acsr*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

<i>ia</i>	(input/output) INTEGER. Array of length $n + 1$, containing indices of elements in the array <i>acsr</i> , such that $ia(I)$ is the index in the array <i>acsr</i> of the first non-zero element from the row I . The value of the last element $ia(n + 1)$ is equal to the number of non-zeros plus one. Refer to <i>rowIndex</i> array description in Sparse Matrix Storage Formats for more details.
<i>acoo</i>	(input/output) REAL for mkl_sscsrcoo. DOUBLE PRECISION for mkl_dscsrcoo. COMPLEX for mkl_ccscsrcoo. DOUBLE COMPLEX for mkl_zscsrcoo. Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>rowind</i>	(input/output) INTEGER. Array of length nnz , contains the row indices for each non-zero element of the matrix A . Refer to <i>rows</i> array description in Coordinate Format for more details.
<i>colind</i>	(input/output) INTEGER. Array of length nnz , contains the column indices for each non-zero element of the matrix A . Refer to <i>columns</i> array description in Coordinate Format for more details.

Output Parameters

<i>nnz</i>	INTEGER. Specifies the number of non-zero element of the matrix A . Refer to <i>nnz</i> description in Coordinate Format for more details.
<i>info</i>	INTEGER. Integer info indicator only for converting the matrix A from the CSR format. If $info=0$, the execution is successful. If $info=1$, the routine is interrupted because there is no space in the arrays <i>acoo</i> , <i>rowind</i> , <i>colind</i> according to the value <i>nzmax</i> .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_sscsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
  INTEGER      job(8)
  INTEGER      n, nnz, info
  INTEGER      ja(*), ia(n+1), rowind(*), colind(*)
  REAL         acsr(*), acoo(*)
```

```
SUBROUTINE mkl_dcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
    INTEGER      job(8)
    INTEGER      n, nnz, info
    INTEGER      ja(*), ia(n+1), rowind(*), colind(*)
    DOUBLE PRECISION      acsr(*), acoo(*)

SUBROUTINE mkl_ccsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
    INTEGER      job(8)
    INTEGER      n, nnz, info
    INTEGER      ja(*), ia(n+1), rowind(*), colind(*)
    COMPLEX      acsr(*), acoo(*)

SUBROUTINE mkl_zcsrcoo(job, n, acsr, ja, ia, nnz, acoo, rowind, colind, info)
    INTEGER      job(8)
    INTEGER      n, nnz, info
    INTEGER      ja(*), ia(n+1), rowind(*), colind(*)
    DOUBLE COMPLEX      acsr(*), acoo(*)
```

C:

```
void mkl_scsrcoo(int *job, int *n, float *acsr, int *ja,
int *ia, int *nnz, float *aco0, int *rowind, int *colind, int *info);
void mkl_dcsrcoo(int *job, int *n, double *acsr, int *ja,
int *ia, int *nnz, double *aco0, int *rowind, int *colind, int *info);
void mkl_ccsrcoo(int *job, int *n, MKL_Complex8 *acsr, int *ja,
int *ia, int *nnz, MKL_Complex8 *aco0, int *rowind, int *colind, int *info);
void mkl_zcsrcoo(int *job, int *n, MKL_Complex16 *acsr, int *ja,
int *ia, int *nnz, MKL_Complex16 *aco0, int *rowind, int *colind, int *info);
```

mkl_?csrbsr

Converts a sparse matrix in the CSR format to the BSR format and vice versa.

Syntax**Fortran:**

```
call mkl_scsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
call mkl_dcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
call mkl_ccsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
call mkl_zcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

C:

```

void mkl_scsrbsr (MKL_INT *job, MKL_INT *m, MKL_INT *mblk, MKL_INT *ldabsr, float
*acsr, MKL_INT *ja, MKL_INT *ia, float *absr, MKL_INT *jab, MKL_INT *iab, MKL_INT
*info);

void mkl_dcsrbsr (MKL_INT *job, MKL_INT *m, MKL_INT *mblk, MKL_INT *ldabsr, double
*acsr, MKL_INT *ja, MKL_INT *ia, double *absr, MKL_INT *jab, MKL_INT *iab, MKL_INT
*info);

void mkl_ccsrbsr (MKL_INT *job, MKL_INT *m, MKL_INT *mblk, MKL_INT *ldabsr,
MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_Complex8 *absr, MKL_INT *jab,
MKL_INT *iab, MKL_INT *info);

void mkl_zcsrbsr (MKL_INT *job, MKL_INT *m, MKL_INT *mblk, MKL_INT *ldabsr,
MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_Complex16 *absr, MKL_INT *jab,
MKL_INT *iab, MKL_INT *info);

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This routine converts a sparse matrix A stored in the compressed sparse row (CSR) format (3-array variation) to the block sparse row (BSR) format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>job</code> <code>job(1)</code> <code>job(2)</code> <code>job(3)</code> <code>job(4)</code> <code>job(6)</code> - job indicator. For conversion to the BSR format: <code>If job(6)=0, only arrays jab, iab are generated for the output storage.</code>	<code>INTEGER</code> Array, contains the following conversion parameters: <code>job(1)</code> If <code>job(1)=0</code> , the matrix in the CSR format is converted to the BSR format; if <code>job(1)=1</code> , the matrix in the BSR format is converted to the CSR format. <code>job(2)</code> If <code>job(2)=0</code> , zero-based indexing for the matrix in CSR format is used; if <code>job(2)=1</code> , one-based indexing for the matrix in CSR format is used. <code>job(3)</code> If <code>job(3)=0</code> , zero-based indexing for the matrix in the BSR format is used; if <code>job(3)=1</code> , one-based indexing for the matrix in the BSR format is used. <code>job(4)</code> is only used for conversion to CSR format. By default, the converter saves the blocks without checking whether an element is zero or not. If <code>job(4)=1</code> , then the converter only saves non-zero elements in blocks.
---	--

If $job(6) > 0$, all output arrays $absr$, jab , and iab are filled in for the output storage.

If $job(6) = -1$, $iab(1)$ returns the number of non-zero blocks.

For conversion to the CSR format:

If $job(6) = 0$, only arrays ja , ia are generated for the output storage.

m

INTEGER. Actual row dimension of the matrix A for convert to the BSR format; block row dimension of the matrix A for convert to the CSR format.

$mblk$

INTEGER. Size of the block in the matrix A .

$ldabsr$

INTEGER. Leading dimension of the array $absr$ as declared in the calling program. $ldabsr$ must be greater than or equal to $mblk * mblk$.

$acsr$

(input/output)

REAL for `mkl_scsrbsr`.

DOUBLE PRECISION for `mkl_dcsrbsr`.

COMPLEX for `mkl_ccsrbsr`.

DOUBLE COMPLEX for `mkl_zcsrbsr`.

Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja

(input/output) INTEGER. Array containing the column indices for each non-zero element of the matrix A .

Its length is equal to the length of the array $acsr$. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia

(input/output) INTEGER. Array of length $m + 1$, containing indices of elements in the array $acsr$, such that $ia(I)$ is the index in the array $acsr$ of the first non-zero element from the row I . The value of the last element $ia(m + 1)$ is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

$absr$

(input/output)

REAL for `mkl_scsrbsr`.

DOUBLE PRECISION for `mkl_dcsrbsr`.

COMPLEX for `mkl_ccsrbsr`.

DOUBLE COMPLEX for `mkl_zcsrbsr`.

Array containing elements of non-zero blocks of the matrix A . Its length is equal to the number of non-zero blocks in the matrix A multiplied by $mblk * mblk$. Refer to *values* array description in [BSR Format](#) for more details.

jab

(input/output) INTEGER. Array containing the column indices for each non-zero block of the matrix A .

Its length is equal to the number of non-zero blocks of the matrix A . Refer to [columns](#) array description in [BSR Format](#) for more details.

iab (input/output) INTEGER. Array of length $(m + 1)$, containing indices of blocks in the array *absr*, such that *iab*(*i*) is the index in the array *absr* of the first non-zero element from the *i*-th row . The value of the last element *iab*($m + 1$) is equal to the number of non-zero blocks plus one. Refer to [rowIndex](#) array description in [BSR Format](#) for more details.

Output Parameters

info INTEGER. Integer info indicator only for converting the matrix A from the CSR format.
 If *info*=0, the execution is successful.
 If *info*=1, it means that *mblk* is equal to 0.
 If *info*=2, it means that *ldabsr* is less than *mblk*mblk* and there is no space for all blocks.

Interfaces

FORTAN 77:

```
SUBROUTINE mkl_scsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

```
    INTEGER      job(8)
```

```
    INTEGER      m, mblk, ldabsr, info
```

```
    INTEGER      ja(*), ia(m+1), jab(*), iab(*)
```

```
    REAL         acsr(*), absr(ldabsr,*)
```

```
SUBROUTINE mkl_dcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

```
    INTEGER      job(8)
```

```
    INTEGER      m, mblk, ldabsr, info
```

```
    INTEGER      ja(*), ia(m+1), jab(*), iab(*)
```

```
    DOUBLE PRECISION      acsr(*), absr(ldabsr,*)
```

```
SUBROUTINE mkl_ccsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

```
    INTEGER      job(8)
```

```
    INTEGER      m, mblk, ldabsr, info
```

```
    INTEGER      ja(*), ia(m+1), jab(*), iab(*)
```

```
    COMPLEX      acsr(*), absr(ldabsr,*)
```

```
SUBROUTINE mkl_zcsrbsr(job, m, mblk, ldabsr, acsr, ja, ia, absr, jab, iab, info)
```

```
    INTEGER      job(8)
```

```
    INTEGER      m, mblk, ldabsr, info
```

```
    INTEGER      ja(*), ia(m+1), jab(*), iab(*)
```

```
    DOUBLE COMPLEX      acsr(*), absr(ldabsr,*)
```

C:

```
void mkl_scsrbsr(int *job, int *m, int *mblk, int *ldabsr, float *acsr, int *ja,
int *ia, float *absr, int *jab, int *iab, int *info);

void mkl_dcsrbsr(int *job, int *m, int *mblk, int *ldabsr, double *acsr, int *ja,
int *ia, double *absr, int *jab, int *iab, int *info);

void mkl_ccsrbsr(int *job, int *m, int *mblk, int *ldabsr, MKL_Complex8 *acsr, int *ja,
int *ia, MKL_Complex8 *absr, int *jab, int *iab, int *info);

void mkl_zcsrbsr(int *job, int *m, int *mblk, int *ldabsr, MKL_Complex16 *acsr, int *ja,
int *ia, MKL_Complex16 *absr, int *jab, int *iab, int *info);
```

mkl_?csrcsc

Converts a square sparse matrix in the CSR format to the CSC format and vice versa.

Syntax**Fortran:**

```
call mkl_sscrcsc(job, m, acsr, ja, ia, acsc, jal, ial, info)
call mkl_dscrcsc(job, m, acsr, ja, ia, acsc, jal, ial, info)
call mkl_ccrcsc(job, m, acsr, ja, ia, acsc, jal, ial, info)
call mkl_zscrcsc(job, m, acsr, ja, ia, acsc, jal, ial, info)
```

C:

```
void mkl_sscrcsc (MKL_INT *job, MKL_INT *m, float *acsr, MKL_INT *ja, MKL_INT *ia,
float *acsc, MKL_INT *jal, MKL_INT *ial, MKL_INT *info);
void mkl_dscrcsc (MKL_INT *job, MKL_INT *m, double *acsr, MKL_INT *ja, MKL_INT *ia,
double *acsc, MKL_INT *jal, MKL_INT *ial, MKL_INT *info);
void mkl_ccrcsc (MKL_INT *job, MKL_INT *m, MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT
*ia, MKL_Complex8 *acsc, MKL_INT *jal, MKL_INT *ial, MKL_INT *info);
void mkl_zscrcsc (MKL_INT *job, MKL_INT *m, MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT
*ia, MKL_Complex16 *acsc, MKL_INT *jal, MKL_INT *ial, MKL_INT *info);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This routine converts a square sparse matrix A stored in the compressed sparse row (CSR) format (3-array variation) to the compressed sparse column (CSC) format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

job

INTEGER

Array, contains the following conversion parameters:

job(1)If *job*(1)=0, the matrix in the CSR format is converted to the CSC format;
if *job*(1)=1, the matrix in the CSC format is converted to the CSR format.*job*(2)If *job*(2)=0, zero-based indexing for the matrix in CSR format is used;
if *job*(2)=1, one-based indexing for the matrix in CSR format is used.*job*(3)If *job*(3)=0, zero-based indexing for the matrix in the CSC format is used;
if *job*(3)=1, one-based indexing for the matrix in the CSC format is used.*job*(6) - job indicator.

For conversion to the CSC format:

If *job*(6)=0, only arrays *ja1*, *ia1* are filled in for the output storage.If *job*(6)≠0, all output arrays *acsc*, *ja1*, and *ia1* are filled in for the output storage.

For conversion to the CSR format:

If *job*(6)=0, only arrays *ja*, *ia* are filled in for the output storage.If *job*(6)≠0, all output arrays *acsr*, *ja*, and *ia* are filled in for the output storage.*m*INTEGER. Dimension of the square matrix *A*.*acsr*

(input/output)

REAL for mkl_scsrcsc.

DOUBLE PRECISION for mkl_dscrcsc.

COMPLEX for mkl_ccsrcsc.

DOUBLE COMPLEX for mkl_zscrcsc.

Array containing non-zero elements of the square matrix *A*. Its length is equal to the number of non-zero elements in the matrix *A*. Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.*ja*(input/output) INTEGER. Array containing the column indices for each non-zero element of the matrix *A*.Its length is equal to the length of the array *acsr*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.*ia*(input/output) INTEGER. Array of length *m* + 1, containing indices of elements in the array *acsr*, such that *ia*(*I*) is the index in the array *acsr* of the first non-zero element from the row *I*. The value of the last element *ia*(*m* + 1) is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

acsc (input/output)
 REAL for mkl_scsrcsc.
 DOUBLE PRECISION for mkl_dcsrcsc.
 COMPLEX for mkl_ccsrcsc.
 DOUBLE COMPLEX for mkl_zcsrcsc.

Array containing non-zero elements of the square matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja1 (input/output) INTEGER. Array containing the row indices for each non-zero element of the matrix A .
 Its length is equal to the length of the array *acsc*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ial (input/output) INTEGER. Array of length $m + 1$, containing indices of elements in the array *acsc*, such that *ial*(I) is the index in the array *acsc* of the first non-zero element from the column I . The value of the last element *ial*($m + 1$) is equal to the number of non-zeros plus one. Refer to *RowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

Output Parameters

info INTEGER. This parameter is not used now.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
```

```
  INTEGER      job(8)
```

```
  INTEGER      m, info
```

```
  INTEGER      ja(*), ia(m+1), ja1(*), ia1(m+1)
```

```
  REAL        acsr(*), acsc(*)
```

```
SUBROUTINE mkl_dcsrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
```

```
  INTEGER      job(8)
```

```
  INTEGER      m, info
```

```
  INTEGER      ja(*), ia(m+1), ja1(*), ia1(m+1)
```

```
  DOUBLE PRECISION    acsr(*), acsc(*)
```

```
SUBROUTINE mkl_ccsrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
```

```
  INTEGER      job(8)
```

```
  INTEGER      m, info
```

```
  INTEGER      ja(*), ia(m+1), ja1(*), ia1(m+1)
```

```
  COMPLEX     acsr(*), acsc(*)
```

```
SUBROUTINE mkl_zscrcsc(job, m, acsr, ja, ia, acsc, ja1, ia1, info)
  INTEGER      job(8)
  INTEGER      m, info
  INTEGER      ja(*), ia(m+1), ja1(*), ia1(m+1)
  DOUBLE COMPLEX      acsr(*), acsc(*)
```

C:

```
void mkl_sscrcsc(int *job, int *m, float *acsr, int *ja,
int *ia, float *acsc, int *jal, int *ial, int *info);

void mkl_dscrcsc(int *job, int *m, double *acsr, int *ja,
int *ia, double *acsc, int *jal, int *ial, int *info);

void mkl_ccsrcsc(int *job, int *m, MKL_Complex8 *acsr, int *ja,
int *ia, MKL_Complex8 *acsc, int *jal, int *ial, int *info);

void mkl_zscrcsc(int *job, int *m, MKL_Complex16 *acsr, int *ja,
int *ia, MKL_Complex16 *acsc, int *jal, int *ial, int *info);
```

mkl_?csrdia

Converts a sparse matrix in the CSR format to the diagonal format and vice versa.

Syntax**Fortran:**

```
call mkl_sscrdia(job, m, acsr, ja, ia, adia, ndiag, distance, iddiag, acsr_rem, ja_rem,
ia_rem, info)

call mkl_dscrdia(job, m, acsr, ja, ia, adia, ndiag, distance, iddiag, acsr_rem, ja_rem,
ia_rem, info)

call mkl_ccsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, iddiag, acsr_rem, ja_rem,
ia_rem, info)

call mkl_zscrdia(job, m, acsr, ja, ia, adia, ndiag, distance, iddiag, acsr_rem, ja_rem,
ia_rem, info)
```

C:

```
void mkl_dscrdia (MKL_INT *job, MKL_INT *n, double *acsr, MKL_INT *ja, MKL_INT *ia,
double *adia, MKL_INT *ndiag, MKL_INT *distance, MKL_INT *iddiag, double *acsr_rem,
MKL_INT *ja_rem, MKL_INT *ia_rem, MKL_INT *info);

void mkl_sscrdia (MKL_INT *job, MKL_INT *n, float *acsr, MKL_INT *ja, MKL_INT *ia,
float *adia, MKL_INT *ndiag, MKL_INT *distance, MKL_INT *iddiag, float *acsr_rem,
MKL_INT *ja_rem, MKL_INT *ia_rem, MKL_INT *info);

void mkl_ccsrdia (MKL_INT *job, MKL_INT *n, MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT
*ia, MKL_Complex8 *adia, MKL_INT *ndiag, MKL_INT *distance, MKL_INT *iddiag,
MKL_Complex8 *acsr_rem, MKL_INT *ja_rem, MKL_INT *ia_rem, MKL_INT *info);
```

```
void mkl_zcsrdia (MKL_INT *job, MKL_INT *n, MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT
*i_a, MKL_Complex16 *adia, MKL_INT *ndiag, MKL_INT *distance, MKL_INT *idiag,
MKL_Complex16 *acsr_rem, MKL_INT *ja_rem, MKL_INT *ia_rem, MKL_INT *info);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This routine converts a sparse matrix A stored in the compressed sparse row (CSR) format (3-array variation) to the diagonal format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<code>job</code> <code>job(1)</code> <code>job(2)</code> <code>job(3)</code> <code>job(6)</code> - job indicator. For conversion to the diagonal format: If <code>job(6)=0</code> , diagonals are not selected internally, and <code>acsr_rem</code> , <code>ja_rem</code> , <code>ia_rem</code> are not filled in for the output storage. If <code>job(6)=1</code> , diagonals are not selected internally, and <code>acsr_rem</code> , <code>ja_rem</code> , <code>ia_rem</code> are filled in for the output storage. If <code>job(6)=10</code> , diagonals are selected internally, and <code>acsr_rem</code> , <code>ja_rem</code> , <code>ia_rem</code> are not filled in for the output storage. If <code>job(6)=11</code> , diagonals are selected internally, and <code>csr_rem</code> , <code>ja_rem</code> , <code>ia_rem</code> are filled in for the output storage. For conversion to the CSR format: If <code>job(6)=0</code> , each entry in the array <code>adia</code> is checked whether it is zero. Zero entries are not included in the array <code>acsr</code> .

If $job(6) \neq 0$, each entry in the array $adia$ is not checked whether it is zero.

m

INTEGER. Dimension of the matrix A .

$acsr$

(input/output)

REAL for mkl_scsrdia.

DOUBLE PRECISION for mkl_dcsrdia.

COMPLEX for mkl_ccsrdia.

DOUBLE COMPLEX for mkl_zcsrdia.

Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to $values$ array description in [Sparse Matrix Storage Formats](#) for more details.

ja

(input/output) INTEGER. Array containing the column indices for each non-zero element of the matrix A .

Its length is equal to the length of the array $acsr$. Refer to $columns$ array description in [Sparse Matrix Storage Formats](#) for more details.

ia

(input/output) INTEGER. Array of length $m + 1$, containing indices of elements in the array $acsr$, such that $ia(I)$ is the index in the array $acsr$ of the first non-zero element from the row I . The value of the last element $ia(m + 1)$ is equal to the number of non-zeros plus one. Refer to $rowIndex$ array description in [Sparse Matrix Storage Formats](#) for more details.

$adia$

(input/output)

REAL for mkl_scsrdia.

DOUBLE PRECISION for mkl_dcsrdia.

COMPLEX for mkl_ccsrdia.

DOUBLE COMPLEX for mkl_zcsrdia.

Array of size $(ndiag \times iddiag)$ containing diagonals of the matrix A .

The key point of the storage is that each element in the array $adia$ retains the row number of the original matrix. To achieve this diagonals in the lower triangular part of the matrix are padded from the top, and those in the upper triangular part are padded from the bottom.

$ndiag$

INTEGER.

Specifies the leading dimension of the array $adia$ as declared in the calling (sub)program, must be at least $\max(1, m)$.

$distance$

INTEGER.

Array of length $iddiag$, containing the distances between the main diagonal and each non-zero diagonal to be extracted. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

$iddiag$

INTEGER.

Number of diagonals to be extracted. For conversion to diagonal format on return this parameter may be modified.

acsr_rem, *ja_rem*, *ia_rem* Remainder of the matrix in the CSR format if it is needed for conversion to the diagonal format.

Output Parameters

info INTEGER. This parameter is not used now.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, iddiag, acsr_rem, ja_rem, ia_rem, info)
```

INTEGER job(8)

INTEGER m, info, ndiag, iddiag

INTEGER ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)

REAL acsr(*), adia(*), acsr_rem(*)

```
SUBROUTINE mkl_dcsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, iddiag, acsr_rem, ja_rem, ia_rem, info)
```

INTEGER job(8)

INTEGER m, info, ndiag, iddiag

INTEGER ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)

DOUBLE PRECISION acsr(*), adia(*), acsr_rem(*)

```
SUBROUTINE mkl_ccsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, iddiag, acsr_rem, ja_rem, ia_rem, info)
```

INTEGER job(8)

INTEGER m, info, ndiag, iddiag

INTEGER ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)

COMPLEX acsr(*), adia(*), acsr_rem(*)

```
SUBROUTINE mkl_zcsrdia(job, m, acsr, ja, ia, adia, ndiag, distance, iddiag, acsr_rem, ja_rem, ia_rem, info)
```

INTEGER job(8)

INTEGER m, info, ndiag, iddiag

INTEGER ja(*), ia(m+1), distance(*), ja_rem(*), ia_rem(*)

DOUBLE COMPLEX acsr(*), adia(*), acsr_rem(*)

C:

```
void mkl_scsrdia(int *job, int *m, float *acsr, int *ja,
int *ia, float *adia, int *ndiag, int *distance, int *distance,
int *iddiag, float *acsr_rem, int *ja_rem, int *ia_rem, int *info);
```

```

void mkl_dcsrdia(int *job, int *m, double *acsr, int *ja,
int *ia, double *adia, int *ndiag, int *distance, int *distance,
int *idiag, double *acsr_rem, int *ja_rem, int *ia_rem, int *info);

void mkl_ccsrdia(int *job, int *m, MKL_Complex8 *acsr, int *ja,
int *ia, MKL_Complex8 *adia, int *ndiag, int *distance, int *distance,
int *idiag, MKL_Complex8 *acsr_rem, int *ja_rem, int *ia_rem, int *info);

void mkl_zcsrdia(int *job, int *m, MKL_Complex16 *acsr, int *ja,
int *ia, MKL_Complex16 *adia, int *ndiag, int *distance, int *distance,
int *idiag, MKL_Complex16 *acsr_rem, int *ja_rem, int *ia_rem, int *info);

```

mkl_?csrsky

Converts a sparse matrix in CSR format to the skyline format and vice versa.

Syntax

Fortran:

```

call mkl_scsrsky(job, m, acsr, ja, ia, asky, pointers, info)
call mkl_dcsrsky(job, m, acsr, ja, ia, asky, pointers, info)
call mkl_ccsrsky(job, m, acsr, ja, ia, asky, pointers, info)
call mkl_zcsrsky(job, m, acsr, ja, ia, asky, pointers, info)

```

C:

```

void mkl_dcsrsky (MKL_INT *job, MKL_INT *m, double *acsr, MKL_INT *ja, MKL_INT *ia,
double *asky, MKL_INT *pointers, MKL_INT *info);

void mkl_scsrsky (MKL_INT *job, MKL_INT *m, float *acsr, MKL_INT *ja, MKL_INT *ia,
float *asky, MKL_INT *pointers, MKL_INT *info);

void mkl_ccsrsky (MKL_INT *job, MKL_INT *m, MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT
*ia, MKL_Complex8 *asky, MKL_INT *pointers, MKL_INT *info);

void mkl_zcsrsky (MKL_INT *job, MKL_INT *m, MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT
*ia, MKL_Complex16 *asky, MKL_INT *pointers, MKL_INT *info);

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This routine converts a sparse matrix A stored in the compressed sparse row (CSR) format (3-array variation) to the skyline format and vice versa.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

job	INTEGER
-----	---------

Array, contains the following conversion parameters:

job(1)

If *job*(1)=0, the matrix in the CSR format is converted to the skyline format;

if *job*(1)=1, the matrix in the skyline format is converted to the CSR format.

job(2)

If *job*(2)=0, zero-based indexing for the matrix in CSR format is used;

if *job*(2)=1, one-based indexing for the matrix in CSR format is used.

job(3)

If *job*(3)=0, zero-based indexing for the matrix in the skyline format is used;

if *job*(3)=1, one-based indexing for the matrix in the skyline format is used.

job(4)

For conversion to the skyline format:

If *job*(4)=0, the upper part of the matrix A in the CSR format is converted.

If *job*(4)=1, the lower part of the matrix A in the CSR format is converted.

For conversion to the CSR format:

If *job*(4)=0, the matrix is converted to the upper part of the matrix A in the CSR format.

If *job*(4)=1, the matrix is converted to the lower part of the matrix A in the CSR format.

job(5)

job(5)=*nzmax* - maximum number of the non-zero elements of the matrix A if *job*(1)=0.

job(6) - job indicator.

Only for conversion to the skyline format:

If *job*(6)=0, only arrays *pointers* is filled in for the output storage.

If *job*(6)=1, all output arrays *asky* and *pointers* are filled in for the output storage.

m

INTEGER. Dimension of the matrix A.

acsr

(input/output)

REAL for mkl_scsrsky.

DOUBLE PRECISION for mkl_dcsrsky.

COMPLEX for mkl_ccsrsky.

DOUBLE COMPLEX for mkl_zcsrsky.

Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

ja

(input/output) INTEGER. Array containing the column indices for each non-zero element of the matrix A .

Its length is equal to the length of the array *acsrr*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia

(input/output) INTEGER. Array of length $m + 1$, containing indices of elements in the array *acsrr*, such that $ia(I)$ is the index in the array *acsrr* of the first non-zero element from the row I . The value of the last element $ia(m + 1)$ is equal to the number of non-zeros plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

asky

(input/output)

REAL for mkl_scsrsky.

DOUBLE PRECISION for mkl_dcsrsky.

COMPLEX for mkl_ccsrsky.

DOUBLE COMPLEX for mkl_zcsrsky.

Array, for a lower triangular part of A it contains the set of elements from each row starting from the first none-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets. Refer to *values* array description in [Skyline Storage Format](#) for more details.

pointers

(input/output) INTEGER.

Array with dimension $(m+1)$, where m is number of rows for lower triangle (columns for upper triangle), $pointers(I) - pointers(1) + 1$ gives the index of element in the array *asky* that is first non-zero element in row (column) I . The value of $pointers(m + 1)$ is set to $nnz + pointers(1)$, where nnz is the number of elements in the array *asky*. Refer to *pointers* array description in [Skyline Storage Format](#) for more details

Output Parameters

info

INTEGER. Integer info indicator only for converting the matrix A from the CSR format.

If $info=0$, the execution is successful.

If $info=1$, the routine is interrupted because there is no space in the array *asky* according to the value *nzmax*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrsky(job, m, acsr, ja, ia, asky, pointers, info)
    INTEGER      job(8)
    INTEGER      m, info
    INTEGER      ja(*), ia(m+1), pointers(m+1)
    REAL         acsr(*), asky(*)

SUBROUTINE mkl_dcsrsky(job, m, acsr, ja, ia, asky, pointers, info)
    INTEGER      job(8)
    INTEGER      m, info
    INTEGER      ja(*), ia(m+1), pointers(m+1)
    DOUBLE PRECISION      acsr(*), asky(*)

SUBROUTINE mkl_ccsrsky(job, m, acsr, ja, ia, asky, pointers, info)
    INTEGER      job(8)
    INTEGER      m, info
    INTEGER      ja(*), ia(m+1), pointers(m+1)
    COMPLEX      acsr(*), asky(*)

SUBROUTINE mkl_zcsrsky(job, m, acsr, ja, ia, asky, pointers, info)
    INTEGER      job(8)
    INTEGER      m, info
    INTEGER      ja(*), ia(m+1), pointers(m+1)
    DOUBLE COMPLEX      acsr(*), asky(*)
```

C:

```
void mkl_scsrsky(int *job, int *m, float *acsr, int *ja,
                  int *ia, float *asky, int *pointers, int *info);

void mkl_dcsrsky(int *job, int *m, double *acsr, int *ja,
                  int *ia, double *asky, int *pointers, int *info);

void mkl_ccsrsky(int *job, int *m, MKL_COMPLEX8 *acsr, int *ja,
                  int *ia, MKL_COMPLEX8 *asky, int *pointers, int *info);

void mkl_zcsrsky(int *job, int *m, MKL_COMPLEX16 *acsr, int *ja,
                  int *ia, MKL_COMPLEX16 *asky, int *pointers, int *info);
```

mkl_?csradd

Computes the sum of two matrices stored in the CSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_scsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic,
nzmax, info)

call mkl_dcsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic,
nzmax, info)

call mkl_ccsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic,
nzmax, info)

call mkl_zcsradd(trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic,
nzmax, info)
```

C:

```
void mkl_scsradd (char *trans, MKL_INT *request, MKL_INT *sort, MKL_INT *m, MKL_INT *n,
float *a, MKL_INT *ja, MKL_INT *ia, float *beta, float *b, MKL_INT *jb, MKL_INT *ib,
float *c, MKL_INT *jc, MKL_INT *ic, MKL_INT *nzmax, MKL_INT *info);

void mkl_dcsradd (char *trans, MKL_INT *request, MKL_INT *sort, MKL_INT *m, MKL_INT *n,
double *a, MKL_INT *ja, MKL_INT *ia, double *beta, double *b, MKL_INT *jb, MKL_INT
*ib, double *c, MKL_INT *jc, MKL_INT *ic, MKL_INT *nzmax, MKL_INT *info);

void mkl_ccsradd (char *trans, MKL_INT *request, MKL_INT *sort, MKL_INT *m, MKL_INT *n,
MKL_Complex8 *a, MKL_INT *ja, MKL_INT *ia, MKL_Complex8 *beta, MKL_Complex8 *b,
MKL_INT *jb, MKL_INT *ib, MKL_Complex8 *c, MKL_INT *jc, MKL_INT *ic, MKL_INT *nzmax,
MKL_INT *info);

void mkl_zcsradd (char *trans, MKL_INT *request, MKL_INT *sort, MKL_INT *m, MKL_INT *n,
MKL_Complex16 *a, MKL_INT *ja, MKL_INT *ia, MKL_Complex16 *beta, MKL_Complex16 *b,
MKL_INT *jb, MKL_INT *ib, MKL_Complex16 *c, MKL_INT *jc, MKL_INT *ic, MKL_INT *nzmax,
MKL_INT *info);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?csradd` routine performs a matrix-matrix operation defined as

$$C := A + \beta \operatorname{op}(B)$$

where:

A, B, C are the sparse matrices in the CSR format (3-array variation).

$\operatorname{op}(B)$ is one of $\operatorname{op}(B) = B$, or $\operatorname{op}(B) = B'$, or $\operatorname{op}(A) = \operatorname{conjg}(B')$

β is a scalar.

The routine works correctly if and only if the column indices in sparse matrix representations of matrices A and B are arranged in the increasing order for each row. If not, use the parameter `sort` (see below) to reorder column indices and the corresponding elements of the input matrices.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>trans</i>	CHARACTER*1. Specifies the operation. If <i>trans</i> = 'N' or 'n', then $C := A + \beta * B$ If <i>trans</i> = 'T' or 't' or 'C' or 'c', then $C := A + \beta * B^T$.
<i>request</i>	INTEGER. If <i>request</i> =0, the routine performs addition. The memory for the output arrays <i>ic</i> , <i>jc</i> , <i>c</i> must be allocated beforehand. If <i>request</i> =1, the routine only computes the values of the array <i>ic</i> of length $m + 1$. The memory for the <i>ic</i> array must be allocated beforehand. On exit the value $ic(m+1) - 1$ is the actual number of the elements in the arrays <i>c</i> and <i>jc</i> . If <i>request</i> =2, after the routine is called previously with the parameter <i>request</i> =1 and after the output arrays <i>jc</i> and <i>c</i> are allocated in the calling program with length at least $ic(m+1) - 1$, the routine performs addition.
<i>sort</i>	INTEGER. Specifies the type of reordering. If this parameter is not set (default), the routine does not perform reordering. If <i>sort</i> =1, the routine arranges the column indices <i>ja</i> for each row in the increasing order and reorders the corresponding values of the matrix <i>A</i> in the array <i>a</i> . If <i>sort</i> =2, the routine arranges the column indices <i>jb</i> for each row in the increasing order and reorders the corresponding values of the matrix <i>B</i> in the array <i>b</i> . If <i>sort</i> =3, the routine performs reordering for both input matrices <i>A</i> and <i>B</i> .
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>a</i>	REAL for mkl_scsradd. DOUBLE PRECISION for mkl_dcsradd. COMPLEX for mkl_ccsradd. DOUBLE COMPLEX for mkl_zcsradd. Array containing non-zero elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i> . For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array a . Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ia INTEGER. Array of length $m + 1$, containing indices of elements in the array a , such that $ia(I)$ is the index in the array a of the first non-zero element from the row I . The value of the last element $ia(m + 1)$ is equal to the number of non-zero elements of the matrix B plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

beta REAL **for** mkl_scsradd.
DOUBLE PRECISION **for** mkl_dcsradd.
COMPLEX **for** mkl_ccsradd.
DOUBLE COMPLEX **for** mkl_zcsradd.
Specifies the scalar *beta*.

b REAL **for** mkl_scsradd.
DOUBLE PRECISION **for** mkl_dcsradd.
COMPLEX **for** mkl_ccsradd.
DOUBLE COMPLEX **for** mkl_zcsradd.
Array containing non-zero elements of the matrix B . Its length is equal to the number of non-zero elements in the matrix B . Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

jb INTEGER. Array containing the column indices for each non-zero element of the matrix B . For each row the column indices must be arranged in the increasing order.
The length of this array is equal to the length of the array *b*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ib INTEGER. Array of length $m + 1$ when *trans* = 'N' or 'n', or $n + 1$ otherwise.
This array contains indices of elements in the array *b*, such that $ib(I)$ is the index in the array *b* of the first non-zero element from the row I . The value of the last element $ib(m + 1)$ or $ib(n + 1)$ is equal to the number of non-zero elements of the matrix B plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

nzmax INTEGER. The length of the arrays *c* and *jc*.
This parameter is used only if *request*=0. The routine stops calculation if the number of elements in the result matrix *C* exceeds the specified value of *nzmax*.

Output Parameters

c REAL **for** mkl_scsradd.
DOUBLE PRECISION **for** mkl_dcsradd.

COMPLEX for `mkl_ccsradd`.

DOUBLE COMPLEX for `mkl_zcsradd`.

Array containing non-zero elements of the result matrix C . Its length is equal to the number of non-zero elements in the matrix C . Refer to `values` array description in [Sparse Matrix Storage Formats](#) for more details.

`jc`

INTEGER. Array containing the column indices for each non-zero element of the matrix C .

The length of this array is equal to the length of the array `c`. Refer to `columns` array description in [Sparse Matrix Storage Formats](#) for more details.

`ic`

INTEGER. Array of length $m + 1$, containing indices of elements in the array `c`, such that $ic(I)$ is the index in the array `c` of the first non-zero element from the row I . The value of the last element $ic(m + 1)$ is equal to the number of non-zero elements of the matrix C plus one. Refer to `rowIndex` array description in [Sparse Matrix Storage Formats](#) for more details.

`info`

INTEGER.

If $info=0$, the execution is successful.

If $info=I>0$, the routine stops calculation in the I -th row of the matrix C because number of elements in C exceeds `nzmax`.

If $info=-1$, the routine calculates only the size of the arrays `c` and `jc` and returns this value plus 1 as the last element of the array `ic`.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax,
info)
```

```
CHARACTER trans
```

```
INTEGER request, sort, m, n, nzmax, info
```

```
INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
```

```
REAL a(*), b(*), c(*), beta
```

```
SUBROUTINE mkl_dcsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax,
info)
```

```
CHARACTER trans
```

```
INTEGER request, sort, m, n, nzmax, info
```

```
INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
```

```
DOUBLE PRECISION a(*), b(*), c(*), beta
```

```
SUBROUTINE mkl_ccsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax,
info)
```

```
CHARACTER trans
INTEGER request, sort, m, n, nzmax, info
INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
COMPLEX a(*), b(*), c(*), beta
```

```
SUBROUTINE mkl_zcsradd( trans, request, sort, m, n, a, ja, ia, beta, b, jb, ib, c, jc, ic, nzmax,
info)
```

```
CHARACTER trans
INTEGER request, sort, m, n, nzmax, info
INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
DOUBLE COMPLEX a(*), b(*), c(*), beta
```

C:

```
void mkl_scsradd(char *trans, int *request, int *sort, int *m, int *n, float *a, int *ja, int *ia,
float *beta, float *b, int *jb, int *ib, float *c,
int *jc, int *ic, int *nzmax, int *info);
```

```
void mkl_dcsradd(char *trans, int *request, int *sort, int *m, int *n,
double *a, int *ja, int *ia, double *beta, double *b, int *jb, int *ib, double *c,
int *jc, int *ic, int *nzmax, int *info);
```

```
void mkl_ccsradd(char *trans, int *request, int *sort, int *m, int *n,
 MKL_Complex8 *a, int *ja, int *ia, MKL_Complex8 *beta, MKL_Complex8 *b,
 int *jb, int *ib, MKL_Complex8 *c, int *jc, int *ic, int *nzmax, int *info);
```

```
void mkl_zcsradd(char *trans, int *request, int *sort, int *m, int *n,
 MKL_Complex16 *a, int *ja, int *ia, MKL_Complex16 *beta, MKL_Complex16 *b,
 int *jb, int *ib, MKL_Complex16 *c, int *jc, int *ic, int *nzmax, int *info);
```

[mkl_?csrmultcsr](#)

Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.

Syntax

Fortran:

```
call mkl_scsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic,
nzmax, info)
```

```
call mkl_dcsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic,
nzmax, info)
```

```
call mkl_ccsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic,
nzmax, info)
```

```
call mkl_zcsrmultcsr(trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic,
nzmax, info)
```

C:

```

void mkl_scsrmultcsr (char *trans, MKL_INT *request, MKL_INT *sort, MKL_INT *m, MKL_INT
*n, MKL_INT *k, float *a, MKL_INT *ja, MKL_INT *ia, float *b, MKL_INT *jb, MKL_INT
*ib, float *c, MKL_INT *jc, MKL_INT *ic, MKL_INT *nzmax, MKL_INT *info);

void mkl_dcsrmultcsr (char *trans, MKL_INT *request, MKL_INT *sort, MKL_INT *m, MKL_INT
*n, MKL_INT *k, double *a, MKL_INT *ja, MKL_INT *ia, double *b, MKL_INT *jb, MKL_INT
*ib, double *c, MKL_INT *jc, MKL_INT *ic, MKL_INT *nzmax, MKL_INT *info);

void mkl_ccsrmultcsr (char *trans, MKL_INT *request, MKL_INT *sort, MKL_INT *m, MKL_INT
*n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ja, MKL_INT *ia, MKL_Complex8 *b, MKL_INT
*jb, MKL_INT *ib, MKL_Complex8 *c, MKL_INT *jc, MKL_INT *ic, MKL_INT *nzmax, MKL_INT
*info);

void mkl_zcsrmultcsr (char *trans, MKL_INT *request, MKL_INT *sort, MKL_INT *m, MKL_INT
*n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ja, MKL_INT *ia, MKL_Complex16 *b, MKL_INT
*jb, MKL_INT *ib, MKL_Complex16 *c, MKL_INT *jc, MKL_INT *ic, MKL_INT *nzmax, MKL_INT
*info);

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?csrmultcsr` routine performs a matrix-matrix operation defined as

$C := \text{op}(A) * B$

where:

A, B, C are the sparse matrices in the CSR format (3-array variation);

$\text{op}(A)$ is one of $\text{op}(A) = A$, or $\text{op}(A) = A'$, or $\text{op}(A) = \text{conjg}(A')$.

You can use the parameter `sort` to perform or not perform reordering of non-zero entries in input and output sparse matrices. The purpose of reordering is to rearrange non-zero entries in compressed sparse row matrix so that column indices in compressed sparse representation are sorted in the increasing order for each row.

The following table shows correspondence between the value of the parameter `sort` and the type of reordering performed by this routine for each sparse matrix involved:

Value of the parameter <code>sort</code>	Reordering of A (arrays a, ja, ia)	Reordering of B (arrays b, ja, ib)	Reordering of C (arrays c, jc, ic)
1	yes	no	yes
2	no	yes	yes
3	yes	yes	yes
4	yes	no	no
5	no	yes	no
6	yes	yes	no
7	no	no	no
arbitrary value not equal to 1, 2,..., 7	no	no	yes

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>trans</i>	CHARACTER*1. Specifies the operation. If <i>trans</i> = 'N' or 'n', then $C := A \times B$ If <i>trans</i> = 'T' or 't' or 'C' or 'c', then $C := A^T \times B$.
<i>request</i>	INTEGER. If <i>request</i> =0, the routine performs multiplication, the memory for the output arrays <i>ic</i> , <i>jc</i> , <i>c</i> must be allocated beforehand. If <i>request</i> =1, the routine computes only values of the array <i>ic</i> of length $m + 1$, the memory for this array must be allocated beforehand. On exit the value $ic(m+1) - 1$ is the actual number of the elements in the arrays <i>c</i> and <i>jc</i> . If <i>request</i> =2, the routine has been called previously with the parameter <i>request</i> =1, the output arrays <i>jc</i> and <i>c</i> are allocated in the calling program and they are of the length $ic(m+1) - 1$ at least.
<i>sort</i>	INTEGER. Specifies whether the routine performs reordering of non-zeros entries in input and/or output sparse matrices (see table above).
<i>m</i>	INTEGER. Number of rows of the matrix <i>A</i> .
<i>n</i>	INTEGER. Number of columns of the matrix <i>A</i> .
<i>k</i>	INTEGER. Number of columns of the matrix <i>B</i> .
<i>a</i>	REAL for mkl_scsrmultcsr. DOUBLE PRECISION for mkl_dcsrmultcsr. COMPLEX for mkl_ccsrmultcsr. DOUBLE COMPLEX for mkl_zcsrmultcsr. Array containing non-zero elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to <i>values</i> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i> . For each row the column indices must be arranged in the increasing order. The length of this array is equal to the length of the array <i>a</i> . Refer to <i>columns</i> array description in Sparse Matrix Storage Formats for more details.
<i>ia</i>	INTEGER. Array of length $m + 1$.

This array contains indices of elements in the array a , such that $ia(I)$ is the index in the array a of the first non-zero element from the row I . The value of the last element $ia(m + 1)$ is equal to the number of non-zero elements of the matrix A plus one. Refer to `rowIndex` array description in [Sparse Matrix Storage Formats](#) for more details.

b REAL **for** `mkl_scsrmultcsr`.

DOUBLE PRECISION **for** `mkl_dcsrmultcsr`.

COMPLEX **for** `mkl_ccsrmultcsr`.

DOUBLE COMPLEX **for** `mkl_zcsrmultcsr`.

Array containing non-zero elements of the matrix B . Its length is equal to the number of non-zero elements in the matrix B . Refer to `values` array description in [Sparse Matrix Storage Formats](#) for more details.

jb INTEGER. Array containing the column indices for each non-zero element of the matrix B . For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array b . Refer to `columns` array description in [Sparse Matrix Storage Formats](#) for more details.

ib INTEGER. Array of length $n + 1$ when $trans = 'N'$ or ' n ', or $m + 1$ otherwise.

This array contains indices of elements in the array b , such that $ib(I)$ is the index in the array b of the first non-zero element from the row I . The value of the last element $ib(n + 1)$ or $ib(m + 1)$ is equal to the number of non-zero elements of the matrix B plus one. Refer to `rowIndex` array description in [Sparse Matrix Storage Formats](#) for more details.

$nzmax$ INTEGER. The length of the arrays c and jc .

This parameter is used only if $request=0$. The routine stops calculation if the number of elements in the result matrix C exceeds the specified value of $nzmax$.

Output Parameters

c REAL **for** `mkl_scsrmultcsr`.

DOUBLE PRECISION **for** `mkl_dcsrmultcsr`.

COMPLEX **for** `mkl_ccsrmultcsr`.

DOUBLE COMPLEX **for** `mkl_zcsrmultcsr`.

Array containing non-zero elements of the result matrix C . Its length is equal to the number of non-zero elements in the matrix C . Refer to `values` array description in [Sparse Matrix Storage Formats](#) for more details.

jc INTEGER. Array containing the column indices for each non-zero element of the matrix C .

The length of this array is equal to the length of the array c . Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ic INTEGER. Array of length $m + 1$ when $trans = 'N'$ or ' n ', or $n + 1$ otherwise.

This array contains indices of elements in the array c , such that $ic(I)$ is the index in the array c of the first non-zero element from the row I . The value of the last element $ic(m + 1)$ or $ic(n + 1)$ is equal to the number of non-zero elements of the matrix C plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

info INTEGER.

If $info=0$, the execution is successful.

If $info=I>0$, the routine stops calculation in the I -th row of the matrix C because number of elements in C exceeds $nzmax$.

If $info=-1$, the routine calculates only the size of the arrays c and jc and returns this value plus 1 as the last element of the array ic .

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax,
                           info)
```

CHARACTER*1 trans

INTEGER request, sort, m, n, k, nzmax, info

INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)

REAL a(*), b(*), c(*)

```
SUBROUTINE mkl_dcsrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax,
                           info)
```

CHARACTER*1 trans

INTEGER request, sort, m, n, k, nzmax, info

INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)

DOUBLE PRECISION a(*), b(*), c(*)

```
SUBROUTINE mkl_ccsrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax,
                           info)
```

CHARACTER*1 trans

INTEGER request, sort, m, n, k, nzmax, info

INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)

COMPLEX a(*), b(*), c(*)

```
SUBROUTINE mkl_zcsrmultcsr( trans, request, sort, m, n, k, a, ja, ia, b, jb, ib, c, jc, ic, nzmax,
info)
```

```
CHARACTER*1 trans
INTEGER request, sort, m, n, k, nzmax, info
INTEGER ja(*), jb(*), jc(*), ia(*), ib(*), ic(*)
DOUBLE COMPLEX a(*), b(*), c(*)
```

C:

```
void mkl_scsrmultcsr(char *trans, int *request, int *sort, int *m, int *n, int *k,
float *a, int *ja, int *ia, float *b, int *jb, int *ib, float *c,
int *jc, int *ic, int *nzmax, int *info);
```

```
void mkl_dcsrmultcsr(char *trans, int *request, int *sort, int *m, int *n, int *k,
double *a, int *ja, int *ia, double *b, int *jb, int *ib, double *c,
int *jc, int *ic, int *nzmax, int *info);
```

```
void mkl_ccsrmultcsr(char *trans, int *request, int *sort, int *m, int *n, int *k,
MKL_Complex8 *a, int *ja, int *ia, MKL_Complex8 *b, int *jb, int *ib,
MKL_Complex8 *c, int *jc, int *ic, int *nzmax, int *info);
```

```
void mkl_zcsrmultcsr(char *trans, int *request, int *sort, int *m, int *n, int *k,
MKL_Complex16 *a, int *ja, int *ia, MKL_Complex16 *b, int *jb, int *ib,
MKL_Complex16 *c, int *jc, int *ic, int *nzmax, int *info);
```

[mkl_?csrmultd](#)

Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing. The result is stored in the dense matrix.

Syntax

Fortran:

```
call mkl_scsrmultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
call mkl_dcsrmultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
call mkl_ccsrmultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
call mkl_zcsrmultd(trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

C:

```
void mkl_dcsrmultd (char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a,
MKL_INT *ja, MKL_INT *ia, double *b, MKL_INT *jb, MKL_INT *ib, double *c, MKL_INT
*ldc);
void mkl_scsrmultd (char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT
*ja, MKL_INT *ia, float *b, MKL_INT *jb, MKL_INT *ib, float *c, MKL_INT *ldc);
void mkl_ccsrmultd (char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a,
MKL_INT *ja, MKL_INT *ia, MKL_Complex8 *b, MKL_INT *jb, MKL_INT *ib, MKL_Complex8 *c,
MKL_INT *ldc);
void mkl_zcsrmultd (char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a,
MKL_INT *ja, MKL_INT *ia, MKL_Complex16 *b, MKL_INT *jb, MKL_INT *ib, MKL_Complex16
*c, MKL_INT *ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?csrmultd` routine performs a matrix-matrix operation defined as

$$C := \text{op}(A) * B$$

where:

A, B are the sparse matrices in the CSR format (3-array variation), C is dense matrix;

$\text{op}(A)$ is one of $\text{op}(A) = A$, or $\text{op}(A) = A'$, or $\text{op}(A) = \text{conjg}(A')$.

The routine works correctly if and only if the column indices in sparse matrix representations of matrices A and B are arranged in the increasing order for each row. If not, use the parameter `sort` (see below) to reorder column indices and the corresponding elements of the input matrices.

NOTE

This routine supports only one-based indexing of the input arrays.

Input Parameters

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

<i>trans</i>	CHARACTER*1. Specifies the operation. If <i>trans</i> = 'N' or 'n', then $C := A * B$ If <i>trans</i> = 'T' or 't' or 'C' or 'c', then $C := A' * B$.
<i>m</i>	INTEGER. Number of rows of the matrix A .
<i>n</i>	INTEGER. Number of columns of the matrix A .
<i>k</i>	INTEGER. Number of columns of the matrix B .
<i>a</i>	REAL for <code>mkl_scsrmultd</code> . DOUBLE PRECISION for <code>mkl_dcsrmultd</code> . COMPLEX for <code>mkl_ccsrmultd</code> . DOUBLE COMPLEX for <code>mkl_zcsrmultd</code> . Array containing non-zero elements of the matrix A . Its length is equal to the number of non-zero elements in the matrix A . Refer to <code>values</code> array description in Sparse Matrix Storage Formats for more details.
<i>ja</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix A . For each row the column indices must be arranged in the increasing order. The length of this array is equal to the length of the array <i>a</i> . Refer to <code>columns</code> array description in Sparse Matrix Storage Formats for more details.

ia INTEGER. Array of length $m + 1$ when $\text{trans} = \text{'N'}$ or 'n' , or $n + 1$ otherwise.

This array contains indices of elements in the array *a*, such that $\text{ia}(I)$ is the index in the array *a* of the first non-zero element from the row I . The value of the last element $\text{ia}(m + 1)$ or $\text{ia}(n + 1)$ is equal to the number of non-zero elements of the matrix *A* plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

b REAL for `mkl_scsrmultd`.

DOUBLE PRECISION for `mkl_dcsrmultd`.

COMPLEX for `mkl_ccsrmultd`.

DOUBLE COMPLEX for `mkl_zcsrmultd`.

Array containing non-zero elements of the matrix *B*. Its length is equal to the number of non-zero elements in the matrix *B*. Refer to *values* array description in [Sparse Matrix Storage Formats](#) for more details.

jb INTEGER. Array containing the column indices for each non-zero element of the matrix *B*. For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array *b*. Refer to *columns* array description in [Sparse Matrix Storage Formats](#) for more details.

ib INTEGER. Array of length $m + 1$.

This array contains indices of elements in the array *b*, such that $\text{ib}(I)$ is the index in the array *b* of the first non-zero element from the row I . The value of the last element $\text{ib}(m + 1)$ is equal to the number of non-zero elements of the matrix *B* plus one. Refer to *rowIndex* array description in [Sparse Matrix Storage Formats](#) for more details.

Output Parameters

c REAL for `mkl_scsrmultd`.

DOUBLE PRECISION for `mkl_dcsrmultd`.

COMPLEX for `mkl_ccsrmultd`.

DOUBLE COMPLEX for `mkl_zcsrmultd`.

Array containing non-zero elements of the result matrix *C*.

ldc INTEGER. Specifies the leading dimension of the dense matrix *C* as declared in the calling (sub)program. Must be at least $\max(m, 1)$ when $\text{trans} = \text{'N'}$ or 'n' , or $\max(1, n)$ otherwise.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_scsrmultd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

```
    CHARACTER*1 trans
```

```
    INTEGER m, n, k, ldc
```

```
    INTEGER ja(*), jb(*), ia(*), ib(*)
```

```
    REAL a(*), b(*), c(ldc, *)
```

```
SUBROUTINE mkl_dcsrmultd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

```
    CHARACTER*1 trans
```

```
    INTEGER m, n, k, ldc
```

```
    INTEGER ja(*), jb(*), ia(*), ib(*)
```

```
    DOUBLE PRECISION a(*), b(*), c(ldc, *)
```

```
SUBROUTINE mkl_ccsrmultd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

```
    CHARACTER*1 trans
```

```
    INTEGER m, n, k, ldc
```

```
    INTEGER ja(*), jb(*), ia(*), ib(*)
```

```
    COMPLEX a(*), b(*), c(ldc, *)
```

```
SUBROUTINE mkl_zcsrmultd( trans, m, n, k, a, ja, ia, b, jb, ib, c, ldc)
```

```
    CHARACTER*1 trans
```

```
    INTEGER m, n, k, ldc
```

```
    INTEGER ja(*), jb(*), ia(*), ib(*)
```

```
    DOUBLE COMPLEX a(*), b(*), c(ldc, *)
```

C:

```
void mkl_scsrmultd(char *trans, int *m, int *n, int *k,
float *a, int *ja, int *ia, float *b, int *jb, int *ib, float *c, int *ldc);
```

```
void mkl_dcsrmultd(char *trans, int *m, int *n, int *k,
double *a, int *ja, int *ia, double *b, int *jb, int *ib, double *c, int *ldc);
```

```
void mkl_ccsrmultd(char *trans, int *m, int *n, int *k,
MKL_Complex8 *a, int *ja, int *ia, MKL_Complex8 *b, int *jb, int *ib,
MKL_Complex8 *c, int *ldc);
```

```
void mkl_zcsrmultd(char *trans, int *m, int *n, int *k,
MKL_Complex16 *a, int *ja, int *ia, MKL_Complex16 *b, int *jb, int *ib,
MKL_Complex16 *c, int *ldc);
```

BLAS-like Extensions

Intel MKL provides C and Fortran routines to extend the functionality of the BLAS routines. These include routines to compute vector products, matrix-vector products, and matrix-matrix products.

Intel MKL also provides routines to perform certain data manipulation, including matrix in-place and out-of-place transposition operations combined with simple matrix arithmetic operations. Transposition operations are Copy As Is, Conjugate transpose, Transpose, and Conjugate. Each routine adds the possibility of scaling during the transposition operation by giving some *alpha* and/or *beta* parameters. Each routine supports both row-major orderings and column-major orderings.

Table “BLAS-like Extensions” lists these routines.

The <?> symbol in the routine short names is a precision prefix that indicates the data type:

<i>s</i>	REAL for Fortran interface, or float for C interface .
<i>d</i>	DOUBLE PRECISION for Fortran interface, or double for C interface .
<i>c</i>	COMPLEX for Fortran interface, or MKL_Complex8 for C interface .
<i>z</i>	DOUBLE COMPLEX for Fortran interface, or MKL_Complex16 for C interface .

BLAS-like Extensions

Routine	Data Types	Description
?axpby ?axpby	<i>s, d, c, z</i>	Scales two vectors, adds them to one another and stores result in the vector (routines)
?gem2vu	<i>s, d</i>	Two matrix-vector products using a general matrix, real data
?gem2vc	<i>c, z</i>	Two matrix-vector products using a general matrix, complex data
?gemm3m	<i>c, z</i>	Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.
mkl_?imatcopy	<i>s, d, c, z</i>	Performs scaling and in-place transposition/copying of matrices.
mkl_?omatcopy	<i>s, d, c, z</i>	Performs scaling and out-of-place transposition/copying of matrices.
mkl_?omatcopy2	<i>s, d, c, z</i>	Performs two-strided scaling and out-of-place transposition/copying of matrices.
mkl_?omatadd	<i>s, d, c, z</i>	Performs scaling and sum of two matrices including their out-of-place transposition/copying.

?axpby

Scales two vectors, adds them to one another and stores result in the vector.

Syntax

FORTRAN 77:

```
call saxpby(n, a, x, incx, b, y, incy)
call daxpby(n, a, x, incx, b, y, incy)
call caxpby(n, a, x, incx, b, y, incy)
call zaxpby(n, a, x, incx, b, y, incy)
```

Fortran 95:

```
call axpby(x, y [,a] [,b])
```

C:

```
void cblas_saxpby (const MKL_INT n, const float a, const float *x, const MKL_INT incx,
const float b, float *y, const MKL_INT incy);

void cblas_daxpby (const MKL_INT n, const double a, const double *x, const MKL_INT incx,
const double b, double *y, const MKL_INT incy);

void cblas_caxpby (const MKL_INT n, const void *a, const void *x, const MKL_INT incx,
const void *b, void *y, const MKL_INT incy);

void cblas_zaxpby (const MKL_INT n, const void *a, const void *x, const MKL_INT incx,
const void *b, void *y, const MKL_INT incy);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?axpby routines perform a vector-vector operation defined as

$$y := a*x + b*y$$

where:

a and *b* are scalars

x and *y* are vectors each with *n* elements.

Input Parameters

<i>n</i>	INTEGER. Specifies the number of elements in vectors <i>x</i> and <i>y</i> .
<i>a</i>	REAL for saxpby DOUBLE PRECISION for daxpby COMPLEX for caxpby DOUBLE COMPLEX for zaxpby Specifies the scalar <i>a</i> .
<i>x</i>	REAL for saxpby DOUBLE PRECISION for daxpby COMPLEX for caxpby DOUBLE COMPLEX for zaxpby Array, size at least $(1 + (n-1) * \text{abs}(incx))$.
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> .
<i>b</i>	REAL for saxpby DOUBLE PRECISION for daxpby COMPLEX for caxpby DOUBLE COMPLEX for zaxpby

Specifies the scalar b .

y REAL for saxpby
 DOUBLE PRECISION for daxpby
 COMPLEX for caxpby
 DOUBLE COMPLEX for zaxpby
 Array, size at least $(1 + (n-1) * \text{abs}(incy))$.

$incy$ INTEGER. Specifies the increment for the elements of y .

Output Parameters

y Contains the updated vector y .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `axpby` interface are the following:

x	Holds the array of size n .
y	Holds the array of size n .
a	The default value is 1.
b	The default value is 1.

?gem2vu

Computes two matrix-vector products using a general matrix (real data)

Syntax

FORTRAN 77:

```
call sgem2vu(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
call dgem2vu(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
```

Fortran 95:

```
call gem2vu(a, x1, x2, y1, y2 [,alpha] [,beta] )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90

Description

The ?gem2vu routines perform two matrix-vector operations defined as

$y1 := \alpha * A * x1 + \beta * y1$,

and

```
y2 := alpha*A'*x2 + beta*y2,
```

where:

alpha and *beta* are scalars,
x1, *x2*, *y1*, and *y2* are vectors,
A is an *m*-by-*n* matrix.

Input Parameters

<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Array, size (<i>lda</i> , <i>n</i>). Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix of coefficients.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, m)$.
<i>x1</i>	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Array, size at least $(1 + (n - 1) * \text{abs}(incx1))$. Before entry, the incremented array <i>x1</i> must contain the vector <i>x1</i> .
<i>incx1</i>	INTEGER. Specifies the increment for the elements of <i>x1</i> . The value of <i>incx1</i> must not be zero.
<i>x2</i>	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Array, size at least $(1 + (m - 1) * \text{abs}(incx2))$. Before entry, the incremented array <i>x2</i> must contain the vector <i>x2</i> .
<i>incx2</i>	INTEGER. Specifies the increment for the elements of <i>x2</i> . The value of <i>incx2</i> must not be zero.
<i>beta</i>	REAL for sgem2vu DOUBLE PRECISION for dgem2vu Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <i>y1</i> and <i>y2</i> need not be set on input.
<i>y1</i>	REAL for sgem2vu

DOUBLE PRECISION for dgem2vu

Array, size at least $(1 + (m - 1) * \text{abs}(incy1))$. Before entry with non-zero *beta*, the incremented array *y1* must contain the vector *y1*.

incy1

INTEGER. Specifies the increment for the elements of *y1*.

The value of *incy1* must not be zero.

y

REAL for sgem2vu

DOUBLE PRECISION for dgem2vu

Array, size at least $(1 + (n - 1) * \text{abs}(incy2))$. Before entry with non-zero *beta*, the incremented array *y2* must contain the vector *y2*.

incy2

INTEGER. Specifies the increment for the elements of *y2*.

The value of *incy2* must not be zero.

Output Parameters

y1 Updated vector *y1*.

y2 Updated vector *y2*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gem2vu` interface are the following:

a Holds the matrix *A* of size (m, n) .

x1 Holds the vector with the number of elements *rx1* where $rx1 = n$.

x2 Holds the vector with the number of elements *rx2* where $rx2 = m$.

y1 Holds the vector with the number of elements *ry1* where $ry1 = m$.

y2 Holds the vector with the number of elements *ry2* where $ry2 = n$.

alpha The default value is 1.

beta The default value is 0.

?gem2vc

Computes two matrix-vector products using a general matrix (complex data)

Syntax

FORTRAN 77:

```
call cgem2vc(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
call zgem2vc(m, n, alpha, a, lda, x1, incx1, x2, incx2, beta, y1, incy1, y2, incy2)
```

Fortran 95:

```
call gem2vc(a, x1, x2, y1, y2 [,alpha] [,beta] )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90

Description

The ?gem2vc routines perform two matrix-vector operations defined as

$y1 := \alpha * A * x1 + \beta * y1,$

and

$y2 := \alpha * \text{conjg}(A') * x2 + \beta * y2,$

where:

α and β are scalars,

$x1, x2, y1$, and $y2$ are vectors,

A is an m -by- n matrix.

Input Parameters

m INTEGER. Specifies the number of rows of the matrix A . The value of m must be at least zero.

n INTEGER. Specifies the number of columns of the matrix A . The value of n must be at least zero.

α COMPLEX for cgem2vc
DOUBLE COMPLEX for zgem2vc

Specifies the scalar α .

a COMPLEX for cgem2vc
DOUBLE COMPLEX for zgem2vc

Array, size (lda, n). Before entry, the leading m -by- n part of the array a must contain the matrix of coefficients.

lda INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of $lدا$ must be at least $\max(1, m)$.

$x1$ COMPLEX for cgem2vc
DOUBLE COMPLEX for zgem2vc
Array, size at least $(1 + (n - 1) * \text{abs}(incx1))$. Before entry, the incremented array $x1$ must contain the vector $x1$.

$incx1$ INTEGER. Specifies the increment for the elements of $x1$.
The value of $incx1$ must not be zero.

$x2$ COMPLEX for cgem2vc
DOUBLE COMPLEX for zgem2vc

Array, size at least $(1 + (m-1) * \text{abs}(incx2))$. Before entry, the incremented array $x2$ must contain the vector $x2$.

incx2 INTEGER. Specifies the increment for the elements of $x2$.

The value of *incx2* must not be zero.

beta COMPLEX for `cgem2vc`

DOUBLE COMPLEX for `zgem2vc`

Specifies the scalar *beta*. When *beta* is set to zero, then $y1$ and $y2$ need not be set on input.

y1 COMPLEX for `cgem2vc`

DOUBLE COMPLEX for `zgem2vc`

Array, size at least $(1 + (m-1) * \text{abs}(incy1))$. Before entry with non-zero *beta*, the incremented array $y1$ must contain the vector $y1$.

incy1 INTEGER. Specifies the increment for the elements of $y1$.

The value of *incy1* must not be zero.

y2 COMPLEX for `cgem2vc`

DOUBLE COMPLEX for `zgem2vc`

Array, size at least $(1 + (n-1) * \text{abs}(incy2))$. Before entry with non-zero *beta*, the incremented array $y2$ must contain the vector $y2$.

incy2 INTEGER. Specifies the increment for the elements of $y2$.

The value of *incy* must not be zero.

INTEGER. Specifies the increment for the elements of y .

Output Parameters

y1 Updated vector $y1$.

y2 Updated vector $y2$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gem2vc` interface are the following:

a Holds the matrix A of size (m, n) .

x1 Holds the vector with the number of elements $rx1$ where $rx1 = n$.

x2 Holds the vector with the number of elements $rx2$ where $rx2 = m$.

y1 Holds the vector with the number of elements $ry1$ where $ry1 = m$.

y2 Holds the vector with the number of elements $ry2$ where $ry2 = n$.

<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 0.

?gemm3m

Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.

Syntax

FORTRAN 77:

```
call cgemm3m(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
call zgemm3m(transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc)
```

Fortran 95:

```
call gemm3m(a, b, c [,transa] [,transb] [,alpha] [,beta])
```

C:

```
void cblas_cgemm3m (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void
*alpha, const void a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void
*beta, void *c, const MKL_INT ldc);

void cblas_zgemm3m (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const
CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void
*alpha, const void a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void
*beta, void *c, const MKL_INT ldc);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: blas.f90
- C: mkl.h

Description

The ?gemm3m routines perform a matrix-matrix operation with general complex matrices. These routines are similar to the ?gemm routines, but they use matrix multiplications(see *Application Notes* below).

The operation is defined as

$$C := \alpha \operatorname{op}(A) \operatorname{op}(B) + \beta C,$$

where:

$\operatorname{op}(x)$ is one of $\operatorname{op}(x) = x$, or $\operatorname{op}(x) = x'$, or $\operatorname{op}(x) = \operatorname{conjg}(x')$,

α and β are scalars,

A , B and C are matrices:

$\operatorname{op}(A)$ is an m -by- k matrix,

$\operatorname{op}(B)$ is a k -by- n matrix,

C is an m -by- n matrix.

Input Parameters

<i>Layout</i>	Specifies whether two-dimensional array storage is row-major (<code>CblasRowMajor</code>) or column-major (<code>CblasColMajor</code>).
<i>transa</i>	CHARACTER*1. Specifies the form of $\text{op}(A)$ used in the matrix multiplication: <code>if transa = 'N' or 'n', then op(A) = A;</code> <code>if transa = 'T' or 't', then op(A) = A';</code> <code>if transa = 'C' or 'c', then op(A) = conjg(A').</code>
<i>transb</i>	CHARACTER*1. Specifies the form of $\text{op}(B)$ used in the matrix multiplication: <code>if transb = 'N' or 'n', then op(B) = B;</code> <code>if transb = 'T' or 't', then op(B) = B';</code> <code>if transb = 'C' or 'c', then op(B) = conjg(B').</code>
<i>m</i>	INTEGER. Specifies the number of rows of the matrix $\text{op}(A)$ and of the matrix C . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix $\text{op}(B)$ and the number of columns of the matrix C . The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. Specifies the number of columns of the matrix $\text{op}(A)$ and the number of rows of the matrix $\text{op}(B)$. The value of <i>k</i> must be at least zero.
<i>alpha</i>	COMPLEX for <code>cgemm3m</code> DOUBLE COMPLEX for <code>zgemm3m</code> Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for <code>cgemm3m</code> DOUBLE COMPLEX for <code>zgemm3m</code> Array, size (lda, ka) , where ka is <i>k</i> when $\text{transa} = 'N'$ or ' <i>n</i> ', and is <i>m</i> otherwise. Before entry with $\text{transa} = 'N'$ or ' <i>n</i> ', the leading <i>m</i> -by- <i>k</i> part of the array <i>a</i> must contain the matrix <i>A</i> , otherwise the leading <i>k</i> -by- <i>m</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When $\text{transa} = 'N'$ or ' <i>n</i> ', then <i>lda</i> must be at least $\max(1, m)$, otherwise <i>lda</i> must be at least $\max(1, k)$.
<i>b</i>	COMPLEX for <code>cgemm3m</code> DOUBLE COMPLEX for <code>zgemm3m</code>

Array, size ldb by kb , where kb is n when $transb = 'N'$ or ' n ', and is k otherwise. Before entry with $transb = 'N'$ or ' n ', the leading k -by- n part of the array b must contain the matrix B , otherwise the leading n -by- k part of the array b must contain the matrix B .

ldb INTEGER. Specifies the leading dimension of b as declared in the calling (sub)program. When $transb = 'N'$ or ' n ', then ldb must be at least $\max(1, k)$, otherwise ldb must be at least $\max(1, n)$.

$beta$ COMPLEX for $cgemm3m$
DOUBLE COMPLEX for $zgemm3m$

Specifies the scalar $beta$.

When $beta$ is equal to zero, then c need not be set on input.

c COMPLEX for $cgemm3m$
DOUBLE COMPLEX for $zgemm3m$
Array, size ldc by n .

Before entry, the leading m -by- n part of the array c must contain the matrix C , except when $beta$ is equal to zero, in which case c need not be set on entry.

ldc INTEGER. Specifies the leading dimension of c as declared in the calling (sub)program. The value of ldc must be at least $\max(1, m)$.

Output Parameters

c Overwritten by the m -by- n matrix $(alpha * op(A) * op(B) + beta * C)$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine $gemm3m$ interface are the following:

a Holds the matrix A of size (ma, ka) where
 $ka = k$ if $transa = 'N'$,
 $ka = m$ otherwise,
 $ma = m$ if $transa = 'N'$,
 $ma = k$ otherwise.

b Holds the matrix B of size (mb, kb) where
 $kb = n$ if $transb = 'N'$,
 $kb = k$ otherwise,
 $mb = k$ if $transb = 'N'$,
 $mb = n$ otherwise.

c Holds the matrix C of size (m, n) .

<i>transa</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>transb</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>alpha</i>	The default value is 1.
<i>beta</i>	The default value is 1.

Application Notes

These routines perform the complex multiplication by forming the real and imaginary parts of the input matrices. It allows to use three real matrix multiplications and five real matrix additions, instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications only gives a 25% reduction of time in matrix operations. This can result in significant savings in computing time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

$$f1(x \text{ op } y) = (x \text{ op } y)(1+\delta), |\delta| \leq u, \text{ op}=\times, /, f1(x \pm y) = x(1+\alpha) \pm y(1+\beta), |\alpha|, |\beta| \leq u$$

then for n -by- n matrix $\hat{C} = f1(C_1 + iC_2) = f1((A_1 + iA_2)(B_1 + iB_2)) = \hat{C}_1 + i\hat{C}_2$ the following estimations are correct

$$\|\hat{C}_1 - C_1\| \leq 2(n+1)u\|A\|_\infty\|B\|_\infty + O(u^2),$$

$$\|\hat{C}_2 - C_2\| \leq 4(n+4)u\|A\|_\infty\|B\|_\infty + O(u^2),$$

where $\|A\|_\infty = \max(\|A_1\|_\infty, \|A_2\|_\infty)$, and $\|B\|_\infty = \max(\|B_1\|_\infty, \|B_2\|_\infty)$.

and hence the matrix multiplications are stable.

mkl_?imatcopy

Performs scaling and in-place transposition/copying of matrices.

Syntax

Fortran:

```
call mkl_simatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)
call mkl_dimatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)
call mkl_cimatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)
call mkl_zimatcopy(ordering, trans, rows, cols, alpha, ab, lda, ldb)
```

C:

```
void mkl_simatcopy (const char ordering, const char trans, size_t rows, size_t cols,
const float alpha, float * AB, size_t lda, size_t ldb);

void mkl_dimatcopy (const char ordering, const char trans, size_t rows, size_t cols,
const double alpha, double * AB, size_t lda, size_t ldb);

void mkl_cimatcopy (const char ordering, const char trans, size_t rows, size_t cols,
const MKL_Complex8 alpha, MKL_Complex8 * AB, size_t lda, size_t ldb);

void mkl_zimatcopy (const char ordering, const char trans, size_t rows, size_t cols,
const MKL_Complex16 alpha, MKL_Complex16 * AB, size_t lda, size_t ldb);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?imatcopy` routine performs scaling and in-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

$AB := \alpha * op(AB)$.

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

NOTE

Different arrays must not overlap.

Input Parameters

<i>ordering</i>	CHARACTER*1. Ordering of the matrix storage. If <i>ordering</i> = 'R' or 'r', the ordering is row-major. If <i>ordering</i> = 'C' or 'c', the ordering is column-major.
<i>trans</i>	CHARACTER*1. Parameter that specifies the operation type. If <i>trans</i> = 'N' or 'n', $op(AB) = AB$ and the matrix AB is assumed unchanged on input. If <i>trans</i> = 'T' or 't', it is assumed that AB should be transposed. If <i>trans</i> = 'C' or 'c', it is assumed that AB should be conjugate transposed. If <i>trans</i> = 'R' or 'r', it is assumed that AB should be only conjugated. If the data is real, then <i>trans</i> = 'R' is the same as <i>trans</i> = 'N', and <i>trans</i> = 'C' is the same as <i>trans</i> = 'T'.
<i>rows</i>	INTEGER. The number of rows in matrix AB before the transpose operation.
<i>cols</i>	INTEGER. The number of columns in matrix AB before the transpose operation.
<i>ab</i>	REAL for <code>mkl_simatcopy</code> . DOUBLE PRECISION for <code>mkl_dimatcopy</code> . COMPLEX for <code>mkl_cimatcopy</code> . DOUBLE COMPLEX for <code>mkl_zimatcopy</code> . Array, size $ab(lda, *)$.
<i>alpha</i>	REAL for <code>mkl_simatcopy</code> . DOUBLE PRECISION for <code>mkl_dimatcopy</code> .

COMPLEX for `mkl_cimatcopy`.

DOUBLE COMPLEX for `mkl_zimatcopy`.

This parameter scales the input matrix by *alpha*.

lda

INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix; measured in the number of elements.

This parameter must be at least *rows* if *ordering* = 'C' or 'c', and $\max(1, \text{cols})$ otherwise.

ldb

INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix; measured in the number of elements.

To determine the minimum value of *ldb* on output, consider the following guideline:

If *ordering* = 'C' or 'c', then

- If *trans* = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, \text{cols})$
- If *trans* = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, \text{rows})$

If *ordering* = 'R' or 'r', then

- If *trans* = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, \text{rows})$
- If *trans* = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, \text{cols})$

Output Parameters

ab

REAL for `mkl_simatcopy`.

DOUBLE PRECISION for `mkl_dimatcopy`.

COMPLEX for `mkl_cimatcopy`.

DOUBLE COMPLEX for `mkl_zimatcopy`.

Array, size *ab*(*ldb*, *).

Contains the matrix *AB*.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_simatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, src_ld, dst_ld
  REAL ab(*), alpha*
```

```
SUBROUTINE mkl_dimatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, src_ld, dst_ld
  DOUBLE PRECISION ab(*), alpha*
```

```
SUBROUTINE mkl_cimatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, src_ld, dst_ld
  COMPLEX ab(*), alpha*
```

```
SUBROUTINE mkl_zimatcopy ( ordering, trans, rows, cols, alpha, ab, lda, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, src_ld, dst_ld
  DOUBLE COMPLEX ab(*), alpha*
```

mkl_?omatcopy

Performs scaling and out-place transposition/copying of matrices.

Syntax

Fortran:

```
call mkl_somatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)
call mkl_domatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)
call mkl_comatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)
call mkl_zomatcopy(ordering, trans, rows, cols, alpha, a, lda, b, ldb)
```

C:

```
void mkl_somatcopy (char ordering, char trans, size_t rows, size_t cols, const float
alpha, const float * A, size_t lda, float * B, size_t ldb);
void mkl_domatcopy (char ordering, char trans, size_t rows, size_t cols, const double
alpha, const double * A, size_t lda, double * B, size_t ldb);
void mkl_comatcopy (char ordering, char trans, size_t rows, size_t cols, const
MKL_Complex8 alpha, const MKL_Complex8 * A, size_t lda, MKL_Complex8 * B, size_t ldb);
void mkl_zomatcopy (char ordering, char trans, size_t rows, size_t cols, const
MKL_Complex16 alpha, const MKL_Complex16 * A, size_t lda, MKL_Complex16 * B, size_t
ldb);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?omatcopy` routine performs scaling and out-of-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

$$B := \alpha * \text{op}(A)$$

The routine parameter descriptions are common for all implemented interfaces with the exception of data types that mostly refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

NOTE

Different arrays must not overlap.

Input Parameters

<i>ordering</i>	CHARACTER*1. Ordering of the matrix storage. If <i>ordering</i> = 'R' or 'r', the ordering is row-major. If <i>ordering</i> = 'C' or 'c', the ordering is column-major.
<i>trans</i>	CHARACTER*1. Parameter that specifies the operation type. If <i>trans</i> = 'N' or 'n', $\text{op}(A)=A$ and the matrix A is assumed unchanged on input. If <i>trans</i> = 'T' or 't', it is assumed that A should be transposed. If <i>trans</i> = 'C' or 'c', it is assumed that A should be conjugate transposed. If <i>trans</i> = 'R' or 'r', it is assumed that A should be only conjugated. If the data is real, then <i>trans</i> = 'R' is the same as <i>trans</i> = 'N', and <i>trans</i> = 'C' is the same as <i>trans</i> = 'T'.
<i>rows</i>	INTEGER. The number of rows in the source matrix.
<i>cols</i>	INTEGER. The number of columns in the source matrix.
<i>alpha</i>	REAL for mkl_somatcopy. DOUBLE PRECISION for mkl_domatcopy. COMPLEX for mkl_comatcopy. DOUBLE COMPLEX for mkl_zomatcopy. This parameter scales the input matrix by <i>alpha</i> .
<i>a</i>	REAL for mkl_somatcopy. DOUBLE PRECISION for mkl_domatcopy. COMPLEX for mkl_comatcopy. DOUBLE COMPLEX for mkl_zomatcopy. Array, size <i>a</i> (<i>lda</i> , *).
<i>lda</i>	INTEGER. (Fortran interface). Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix; measured in the number of elements. This parameter must be at least $\max(1, \text{rows})$ if <i>ordering</i> = 'C' or 'c', and $\max(1, \text{cols})$ otherwise.
<i>b</i>	REAL for mkl_somatcopy. DOUBLE PRECISION for mkl_domatcopy. COMPLEX for mkl_comatcopy. DOUBLE COMPLEX for mkl_zomatcopy. Array, size <i>b</i> (<i>ldb</i> , *).

ldb

INTEGER. (Fortran interface). Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix; measured in the number of elements.

To determine the minimum value of *ldb* on output, consider the following guideline:

If *ordering* = 'C' or 'c', then

- If *trans* = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, \text{cols})$
- If *trans* = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, \text{rows})$

If *ordering* = 'R' or 'r', then

- If *trans* = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, \text{rows})$
- If *trans* = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, \text{cols})$

Output Parameters

b

REAL for mkl_somatcopy.

DOUBLE PRECISION for mkl_damatcopy.

COMPLEX for mkl_comatcopy.

DOUBLE COMPLEX for mkl_zomatcopy.

Array, size at least *m*.

Contains the destination matrix.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_somatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, ldb
  REAL alpha, b(ldb,*), a(lda,*)
```

```
SUBROUTINE mkl_damatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, ldb
  DOUBLE PRECISION alpha, b(ldb,*), a(lda,*)
```

```
SUBROUTINE mkl_comatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, ldb
  COMPLEX alpha, b(ldb,*), a(lda,*)
```

```
SUBROUTINE mkl_zomatcopy ( ordering, trans, rows, cols, alpha, a, lda, b, ldb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, ldb
  DOUBLE COMPLEX alpha, b(ldb,*), a(lda,*)
```

mkl_?omatcopy2

Performs two-strided scaling and out-of-place transposition/copying of matrices.

Syntax

Fortran:

```
call mkl_somatcopy2(ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb,
strideb)

call mkl_domatcopy2(ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb,
strideb)

call mkl_comatcopy2(ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb,
strideb)

call mkl_zomatcopy2(ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb,
strideb)
```

C:

```
void mkl_somatcopy2 (char ordering, char trans, size_t rows, size_t cols, const float
alpha, const float * A, size_t lda, size_t stridea, float * B, size_t ldb, size_t
strideb);

void mkl_domatcopy2 (char ordering, char trans, size_t rows, size_t cols, const double
alpha, const double * A, size_t lda, size_t stridea, double * B, size_t ldb, size_t
strideb);

void mkl_comatcopy2 (char ordering, char trans, size_t rows, size_t cols, const
MKL_Complex8 alpha, const MKL_Complex8 * A, size_t lda, size_t stridea, MKL_Complex8 *
B, size_t ldb, size_t strideb);

void mkl_zomatcopy2 (char ordering, char trans, size_t rows, size_t cols, const
MKL_Complex16 alpha, const MKL_Complex16 * A, size_t lda, size_t stridea, MKL_Complex16
* B, size_t ldb, size_t strideb);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?omatcopy2` routine performs two-strided scaling and out-of-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

$$B := \text{alpha}^*\text{op}(A)$$

Normally, matrices in the BLAS or LAPACK are specified by a single stride index. For instance, in the column-major order, $A(2,1)$ is stored in memory one element away from $A(1,1)$, but $A(1,2)$ is a leading dimension away. The leading dimension in this case is the single stride. If a matrix has two strides, then both $A(2,1)$ and $A(1,2)$ may be an arbitrary distance from $A(1,1)$.

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

NOTE

Different arrays must not overlap.

Input Parameters

<i>ordering</i>	CHARACTER*1. Ordering of the matrix storage. If <i>ordering</i> = 'R' or 'r', the ordering is row-major. If <i>ordering</i> = 'C' or 'c', the ordering is column-major.
<i>trans</i>	CHARACTER*1. Parameter that specifies the operation type. If <i>trans</i> = 'N' or 'n', $\text{op}(A) = A$ and the matrix A is assumed unchanged on input. If <i>trans</i> = 'T' or 't', it is assumed that A should be transposed. If <i>trans</i> = 'C' or 'c', it is assumed that A should be conjugate transposed. If <i>trans</i> = 'R' or 'r', it is assumed that A should be only conjugated. If the data is real, then <i>trans</i> = 'R' is the same as <i>trans</i> = 'N', and <i>trans</i> = 'C' is the same as <i>trans</i> = 'T'.
<i>rows</i>	INTEGER. The number of matrix rows.
<i>cols</i>	INTEGER. The number of matrix columns.
<i>alpha</i>	REAL for mkl_somatcopy2. DOUBLE PRECISION for mkl_domatcopy2. COMPLEX for mkl_comatcopy2. DOUBLE COMPLEX for mkl_zomatcopy2. This parameter scales the input matrix by <i>alpha</i> .
<i>a</i>	REAL for mkl_somatcopy2. DOUBLE PRECISION for mkl_domatcopy2. COMPLEX for mkl_comatcopy2. DOUBLE COMPLEX for mkl_zomatcopy2. Array, size <i>a</i> (*).
<i>lda</i>	INTEGER. Distance between the first elements in adjacent rows in the source matrix; measured in the number of elements. This parameter must be at least $\max(1, \text{rows})$ if <i>ordering</i> = 'C' or 'c', and $\max(1, \text{cols})$ otherwise.
<i>stridea</i>	INTEGER. Distance between the first elements in adjacent columns (for row-major layout) or between the first elements in adjacent rows (for column-major layout) in the source matrix; measured in the number of elements. This parameter must be at least $\max(1, \text{cols})$.

b

REAL for mkl_somatcopy2.
 DOUBLE PRECISION for mkl_domatcopy2.
 COMPLEX for mkl_comatcopy2.
 DOUBLE COMPLEX for mkl_zomatcopy2.
 Array, size *b*(*).

ldb
 INTEGER. Distance between the first elements in adjacent rows in the destination matrix; measured in the number of elements.

To determine the minimum value of *ldb* on output, consider the following guideline:

- If *trans* = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, \text{rows})$
- If *trans* = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, \text{cols})$

strideb
 INTEGER. Distance between the first elements in adjacent columns in the destination matrix; measured in the number of elements.

To determine the minimum value of *strideb* on output, consider the following guideline:

- If *trans* = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, \text{cols})$
- If *trans* = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, \text{rows})$

Output Parameters

b

REAL for mkl_somatcopy2.
 DOUBLE PRECISION for mkl_domatcopy2.
 COMPLEX for mkl_comatcopy2.
 DOUBLE COMPLEX for mkl_zomatcopy2.
 Array, size at least *m*.

Contains the destination matrix.

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_somatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, stridea, ldb, strideb
  REAL alpha, b(*), a(*)
```

```
SUBROUTINE mkl_domatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, stridea, ldb, strideb
  DOUBLE PRECISION alpha, b(*), a(*)
```

```
SUBROUTINE mkl_comatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, stridea, ldb, strideb
  COMPLEX alpha, b(*), a(*)
```

```
SUBROUTINE mkl_zomatcopy2 ( ordering, trans, rows, cols, alpha, a, lda, stridea, b, ldb, strideb )
  CHARACTER*1 ordering, trans
  INTEGER rows, cols, lda, stridea, ldb, strideb
  DOUBLE COMPLEX alpha, b(*), a(*)
```

[mkl_?omatadd](#)

Performs scaling and sum of two matrices including their out-of-place transposition/copying.

Syntax

Fortran:

```
call mkl_somatadd(ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc)
call mkl_domatadd(ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc)
call mkl_comatadd(ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc)
call mkl_zomatadd(ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc)
```

C:

```
void mkl_somatadd (char ordering, char transa, char transb, size_t m, size_t n, const
float alpha, const float * A, size_t lda, const float beta, const float * B, size_t
ldb, float * C, size_t ldc);

void mkl_domatadd (char ordering, char transa, char transb, size_t m, size_t n, const
double alpha, const double * A, size_t lda, const double beta, const double * B,
size_t ldb, double * C, size_t ldc);

void mkl_comatadd (char ordering, char transa, char transb, size_t m, size_t n, const
MKL_Complex8 alpha, const MKL_Complex8 * A, size_t lda, const MKL_Complex8 beta, const
MKL_Complex8 * B, size_t ldb, MKL_Complex8 * C, size_t ldc);

void mkl_zomatadd (char ordering, char transa, char transb, size_t m, size_t n, const
MKL_Complex16 alpha, const MKL_Complex16 * A, size_t lda, const MKL_Complex16 beta,
const MKL_Complex16 * B, size_t ldb, MKL_Complex16 * C, size_t ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `mkl_?omatadd` routine scaling and sum of two matrices including their out-of-place transposition/copying. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The following out-of-place memory movement is done:

$$C := \alpha \cdot op(A) + \beta \cdot op(B)$$

`op(A)` is either transpose, conjugate-transpose, or leave alone depending on `transa`. If no transposition of the source matrices is required, `m` is the number of rows and `n` is the number of columns in the source matrices `A` and `B`. In this case, the output matrix `C` is `m`-by-`n`.

Parameter descriptions are common for all implemented interfaces with the exception of data types that refer here to the FORTRAN 77 standard types. Data types specific to the different interfaces are described in the section "**Interfaces**" below.

NOTE

Note that different arrays must not overlap.

Input Parameters

<i>ordering</i>	CHARACTER*1. Ordering of the matrix storage. If <i>ordering</i> = 'R' or 'r', the ordering is row-major. If <i>ordering</i> = 'C' or 'c', the ordering is column-major.
<i>transa</i>	CHARACTER*1. Parameter that specifies the operation type on matrix A. If <i>transa</i> = 'N' or 'n', $\text{op}(A)=A$ and the matrix A is assumed unchanged on input. If <i>transa</i> = 'T' or 't', it is assumed that A should be transposed. If <i>transa</i> = 'C' or 'c', it is assumed that A should be conjugate transposed. If <i>transa</i> = 'R' or 'r', it is assumed that A should be only conjugated. If the data is real, then <i>transa</i> = 'R' is the same as <i>transa</i> = 'N', and <i>transa</i> = 'C' is the same as <i>transa</i> = 'T'.
<i>transb</i>	CHARACTER*1. Parameter that specifies the operation type on matrix B. If <i>transb</i> = 'N' or 'n', $\text{op}(B)=B$ and the matrix B is assumed unchanged on input. If <i>transb</i> = 'T' or 't', it is assumed that B should be transposed. If <i>transb</i> = 'C' or 'c', it is assumed that B should be conjugate transposed. If <i>transb</i> = 'R' or 'r', it is assumed that B should be only conjugated. If the data is real, then <i>transb</i> = 'R' is the same as <i>transb</i> = 'N', and <i>transb</i> = 'C' is the same as <i>transb</i> = 'T'.
<i>m</i>	INTEGER. The number of matrix rows.
<i>n</i>	INTEGER. The number of matrix columns.
<i>alpha</i>	REAL for mkl_somatadd. DOUBLE PRECISION for mkl_domatadd. COMPLEX for mkl_comatadd. DOUBLE COMPLEX for mkl_zomatadd. This parameter scales the input matrix by <i>alpha</i> .
<i>a</i>	REAL for mkl_somatadd. DOUBLE PRECISION for mkl_domatadd.

COMPLEX **for** mkl_comatadd.

DOUBLE COMPLEX **for** mkl_zomatadd.

Array, size $a(1da, *)$.

lda INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix A ; measured in the number of elements.

This parameter must be at least $\max(1, \text{rows})$ if $\text{ordering} = 'C'$ or ' c ', and $\max(1, \text{cols})$ otherwise.

beta REAL **for** mkl_somatadd.

DOUBLE PRECISION **for** mkl_domatadd.

COMPLEX **for** mkl_comatadd.

DOUBLE COMPLEX **for** mkl_zomatadd.

This parameter scales the input matrix by *beta*.

b REAL **for** mkl_somatadd.

DOUBLE PRECISION **for** mkl_domatadd.

COMPLEX **for** mkl_comatadd.

DOUBLE COMPLEX **for** mkl_zomatadd.

Array, size $b(1db, *)$.

ldb INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix B ; measured in the number of elements.

This parameter must be at least $\max(1, \text{rows})$ if $\text{ordering} = 'C'$ or ' c ', and $\max(1, \text{cols})$ otherwise.

Output Parameters

c REAL **for** mkl_somatadd.

DOUBLE PRECISION **for** mkl_domatadd.

COMPLEX **for** mkl_comatadd.

DOUBLE COMPLEX **for** mkl_zomatadd.

Array, size $c(1dc, *)$.

ldc INTEGER. Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix C ; measured in the number of elements.

To determine the minimum value of *ldc*, consider the following guideline:

If $\text{ordering} = 'C'$ or ' c ', then

- If transa or $\text{transb} = 'T'$ or ' t ' or ' C ' or ' c ', this parameter must be at least $\max(1, \text{cols})$
- If transa or $\text{transb} = 'N'$ or ' n ' or ' R ' or ' r ', this parameter must be at least $\max(1, \text{rows})$

If *ordering* = 'R' or 'r', then

- If *transa* or *transb* = 'T' or 't' or 'C' or 'c', this parameter must be at least $\max(1, \text{rows})$
- If *transa* or *transb* = 'N' or 'n' or 'R' or 'r', this parameter must be at least $\max(1, \text{cols})$

Interfaces

FORTRAN 77:

```
SUBROUTINE mkl_somatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )
  CHARACTER*1 ordering, transa, transb
  INTEGER m, n, lda, ldb, ldc
  REAL alpha, beta
  REAL a(lda,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_damatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )
  CHARACTER*1 ordering, transa, transb
  INTEGER m, n, lda, ldb, ldc
  DOUBLE PRECISION alpha, beta
  DOUBLE PRECISION a(lda,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_comatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )
  CHARACTER*1 ordering, transa, transb
  INTEGER m, n, lda, ldb, ldc
  COMPLEX alpha, beta
  COMPLEX a(lda,*), b(ldb,*), c(ldc,*)
```

```
SUBROUTINE mkl_zomatadd ( ordering, transa, transb, m, n, alpha, a, lda, beta, b, ldb, c, ldc )
  CHARACTER*1 ordering, transa, transb
  INTEGER m, n, lda, ldb, ldc
  DOUBLE COMPLEX alpha, beta
  DOUBLE COMPLEX a(lda,*), b(ldb,*), c(ldc,*)
```

LAPACK Routines

This chapter describes the Intel® Math Kernel Library implementation of routines from the LAPACK package that are used for solving systems of linear equations, linear least squares problems, eigenvalue and singular value problems, and performing a number of related computational tasks. The library includes LAPACK routines for both real and complex data. Routines are supported for systems of equations with the following types of matrices:

- general
- banded
- symmetric or Hermitian positive-definite (full, packed, and rectangular full packed (RFP) storage)
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian indefinite (both full and packed storage)
- symmetric or Hermitian indefinite banded
- triangular (full, packed, and RFP storage)
- triangular banded
- tridiagonal
- diagonally dominant tridiagonal.

NOTE

Different arrays used as parameters to Intel MKL LAPACK routines must not overlap.

WARNING

LAPACK routines assume that input matrices do not contain IEEE 754 special values such as `INF` or `NaN` values. Using these special values may cause LAPACK to return unexpected results or become unstable.

Intel MKL supports the Fortran 95 interface, which uses simplified routine calls with shorter argument lists, in addition to the FORTRAN 77 interface to LAPACK computational and driver routines. The syntax section of the routine description gives the calling sequence for the Fortran 95 interface, where available, immediately after the FORTRAN 77 calls.

Routine Naming Conventions

To call one of the routines from a FORTRAN 77 program, you can use the LAPACK name.

LAPACK names have the structure `?yyzzz` or `?yyzz`, where the initial symbol `?` indicates the data type:

s	real, single precision
c	complex, single precision
d	real, double precision
z	complex, double precision

Some routines can have combined character codes, such as `ds` or `zc`.

The Fortran 95 interfaces to the LAPACK computational and driver routines are the same as the FORTRAN 77 names but without the first letter that indicates the data type. For example, the name of the routine that performs a triangular factorization of general real matrices in Fortran 95 is `getrf`. Different data types are handled through the definition of a specific internal parameter that refers to a module block with named constants for single and double precision.

C Interface Conventions

The C interfaces are implemented for most of the Intel MKL LAPACK driver and computational routines.

The arguments of the C interfaces for the Intel MKL LAPACK functions comply with the following rules:

- Scalar input arguments are passed by value.
- Array arguments are passed by reference.
- Array input arguments are declared with the `const` modifier.
- Function arguments are passed by pointer.
- An integer return value replaces the `info` output parameter. The return value equal to 0 means the function operation is completed successfully. See also [special error codes](#) below.

Function Prototypes

Some Intel MKL functions differ in data types they support and vary in the parameters they take.

Each function type has a unique prototype defined. Use this prototype when you call the function from your application program. In most cases, Intel MKL supports four distinct floating-point precisions. Each corresponding prototype looks similar, usually differing only in the data type. To avoid listing all the prototypes in every supported precision, a generic prototype template is provided.

`<?>` denotes precision and is `s`, `d`, `c`, or `z`:

- `s` for real, single precision
- `d` for real, double precision
- `c` for complex, single precision
- `z` for complex, double precision

`<datatype>` stands for a respective data type: `float`, `double`, `lapack_complex_float`, or `lapack_complex_double`.

For example, the C prototype template for the `?pptrs` function that solves a system of linear equations with a packed Cholesky-factored symmetric (Hermitian) positive-definite matrix looks as follows:

```
lapack_int LAPACKE_<?>pptrs(int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
                           const <datatype>* ap, <datatype>* b, lapack_int ldb);
```

To obtain the function name and parameter list that corresponds to a specific precision, replace the `<?>` symbol with `s`, `d`, `c`, or `z` and the `<datatype>` field with the corresponding data type (`float`, `double`, `lapack_complex_float`, or `lapack_complex_double` respectively).

A specific example follows. To solve a system of linear equations with a packed Cholesky-factored Hermitian positive-definite matrix with complex precision, use the following:

```
lapack_int LAPACKE_cpptrs(int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
                           const lapack_complex_float* ap, lapack_complex_float* b, lapack_int ldb);
```

NOTE

For the `select` parameter, the respective values of the `<datatype>` field for `s`, `d`, `c`, or `z` are as follows: `LAPACK_S_SELECT3`, `LAPACK_D_SELECT3`, `LAPACK_C_SELECT2`, and `LAPACK_Z_SELECT2`.

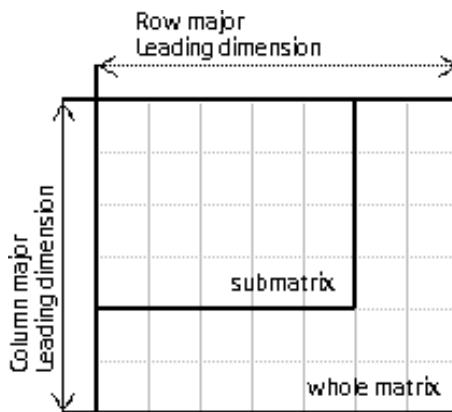
Matrix Layout

Most of the LAPACK C interfaces have an additional parameter `matrix_layout` of type `int` as their first argument. This parameter specifies whether the two-dimensional arrays are row-major (`LAPACK_ROW_MAJOR`) or column-major (`LAPACK_COL_MAJOR`).

In general the leading dimension `lда` is equal to the number of elements in the major dimension. It is also equal to the distance in elements between two neighboring elements in a line in the minor dimension. If there are no extra elements in a matrix with m rows and n columns, then

- For row-major ordering: the number of elements in a row is n , and row i is stored in memory right after row $i-1$. Therefore `lда` is n .
- For column-major ordering: the number of elements in a column is m , and column i is stored in memory right after column $i-1$. Therefore `lда` is m .

To refer to a submatrix with dimensions k by l , use the number of elements in the major dimension of the whole matrix (as above) as the leading dimension and k and l in the subroutine's input parameters to describe the size of the submatrix.



Workspace Arrays

In general, the LAPACK C interface omits workspace parameters because workspace is allocated during runtime and released upon completion of the function operation.

For some functions, `work` arrays contain valuable information on exit. In such cases, the interface contains an additional argument or arguments, namely:

- `?gesvx` and `?gbsvx` contain `rpivot`
- `?gesvd` contains `superb`
- `?gejsv` and `?gesvj` contain `istat` and `stat`, respectively.

If you prefer to allocate workspace functions yourself, the LAPACK C interface provides alternate interfaces with `work` parameters. The name of the alternate interface is the same as the LAPACK C interface with `_work` appended. For example, the syntax for the singular value decomposition of a real bidiagonal matrix is:

FORTRAN 77: `call sbdsdc (uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)`

C LAPACK interface: `lapack_int LAPACKE_sbdsdc (int matrix_layout, char uplo, char compq, lapack_int n, float* d, float* e, float* u, lapack_int ldu, float* vt, lapack_int ldvt, float* q, lapack_int* iq);`

Alternate C LAPACK interface with *work* parameters:

```
lapack_int LAPACKE_sbdsdc_work( int matrix_layout, char uplo,
                                char compq, lapack_int n, float* d, float* e, float* u,
                                lapack_int ldu, float* vt, lapack_int ldvt, float* q, lapack_int* iq,
                                float* work, lapack_int* iwork );
```

Function Types

The function types are used in non-symmetric eigenproblem functions only.

```
typedef lapack_logical (*LAPACK_S_SELECT2) (const float*, const float*);
typedef lapack_logical (*LAPACK_S_SELECT3) (const float*, const float*, const float*);
typedef lapack_logical (*LAPACK_D_SELECT2) (const double*, const double*);
typedef lapack_logical (*LAPACK_D_SELECT3) (const double*, const double*, const double*);

typedef lapack_logical (*LAPACK_C_SELECT1) (const lapack_complex_float*);
typedef lapack_logical (*LAPACK_C_SELECT2) (const lapack_complex_float*, const lapack_complex_float*);
typedef lapack_logical (*LAPACK_Z_SELECT1) (const lapack_complex_double*);
typedef lapack_logical (*LAPACK_Z_SELECT2) (const lapack_complex_double*, const lapack_complex_double*);
```

Mapping FORTRAN Data Types against C Data Types

FORTRAN Data Types vs. C Data Types

FORTRAN	C
INTEGER	lapack_int
LOGICAL	lapack_logical
REAL	float
DOUBLE PRECISION	double
COMPLEX	lapack_complex_float
COMPLEX*16/DOUBLE COMPLEX	lapack_complex_double
CHARACTER	char

C Type Definitions

```
#ifndef lapack_int
#define lapack_int MKL_INT
#endif

#ifndef lapack_logical
#define lapack_logical lapack_int
#endif
```

Complex Type Definitions

Complex type for single precision:

```
#ifndef lapack_complex_float
#define lapack_complex_float MKL_Complex8
#endif
```

Complex type for double precision:

```
#ifndef lapack_complex_double
#define lapack_complex_double MKL_Complex16
#endif
```

Matrix Order Definitions

```
#define LAPACK_ROW_MAJOR 101
#define LAPACK_COL_MAJOR 102
```

See [Matrix Order](#) for an explanation of row-major order and column-major order storage.

Error Code Definitions

```
#define LAPACK_WORK_MEMORY_ERROR -1010 /* Failed to allocate memory
                                         for a working array */
#define LAPACK_TRANSPOSE_MEMORY_ERROR -1011 /* Failed to allocate memory
                                         for transposed matrix */
```

If the return value is `-i`, the `-i`-th parameter has an invalid value.

Fortran 95 Interface Conventions

Intel® MKL implements the Fortran 95 interface to LAPACK through wrappers that call respective FORTRAN 77 routines. This interface uses such Fortran 95 features as assumed-shape arrays and optional arguments to provide simplified calls to LAPACK routines with fewer arguments.

NOTE

For LAPACK, Intel MKL offers two types of the Fortran 95 interfaces:

- using `mkl_lapack.fi` only through the `include 'mkl_lapack.fi'` statement. Such interfaces allow you to make use of the original LAPACK routines with all their arguments
- using `lapack.f90` that includes improved interfaces. This file is used to generate the module files `lapack95.mod` and `f95_precision.mod`. See also the section "Fortran 95 interfaces and wrappers to LAPACK and BLAS" of the *Intel® MKL User's Guide* for details. The module files are used to process the FORTRAN use clauses referencing the LAPACK interface: `use lapack95` and `use f95_precision`.

The main conventions for the Fortran 95 interface are as follows:

- The names of arguments used in Fortran 95 call are typically the same as for the respective generic (FORTRAN 77) interface. In rare cases, formal argument names may be different. For instance, `select` instead of `selctg`.
- Input arguments such as array dimensions are not required in Fortran 95 and are skipped from the calling sequence. Array dimensions are reconstructed from the user data that must exactly follow the required array shape.

Another type of generic arguments that are skipped in the Fortran 95 interface are arguments that represent workspace arrays (such as `work`, `rwork`, and so on). The only exception are cases when workspace arrays return significant information on output.

An argument can also be skipped if its value is completely defined by the presence or absence of another argument in the calling sequence, and the restored value is the only meaningful value for the skipped argument.

- Some generic arguments are declared as optional in the Fortran 95 interface and may or may not be present in the calling sequence. An argument can be declared optional if it meets one of the following conditions:
 - If an argument value is completely defined by the presence or absence of another argument in the calling sequence, it can be declared optional. The difference from the skipped argument in this case is that the optional argument can have some meaningful values that are distinct from the value reconstructed by default. For example, if some argument (like `jobz`) can take only two values and one of these values directly implies the use of another argument, then the value of `jobz` can be uniquely reconstructed from the actual presence or absence of this second argument, and `jobz` can be omitted.

- If an input argument can take only a few possible values, it can be declared as optional. The default value of such argument is typically set as the first value in the list and all exceptions to this rule are explicitly stated in the routine description.
- If an input argument has a natural default value, it can be declared as optional. The default value of such optional argument is set to its natural default value.
- Argument *info* is declared as optional in the Fortran 95 interface. If it is present in the calling sequence, the value assigned to *info* is interpreted as follows:
 - If this value is more than -1000, its meaning is the same as in the FORTRAN 77 routine.
 - If this value is equal to -1000, it means that there is not enough work memory.
 - If this value is equal to -1001, incompatible arguments are present in the calling sequence.
 - If this value is equal to *i*, the *i*th parameter (counting parameters in the FORTRAN 77 interface, not the Fortran 95 interface) had an illegal value.
- Optional arguments are given in square brackets in the Fortran 95 call syntax.

The "Fortran 95 Notes" subsection at the end of the topic describing each routine details concrete rules for reconstructing the values of the omitted optional parameters.

Intel® MKL Fortran 95 Interfaces for LAPACK Routines vs. Netlib Implementation

The following list presents general digressions of the Intel MKL LAPACK95 implementation from the Netlib analog:

- The Intel MKL Fortran 95 interfaces are provided for pure procedures.
- Names of interfaces do not contain the `LA_` prefix.
- An optional array argument always has the `target` attribute.
- Functionality of the Intel MKL LAPACK95 wrapper is close to the FORTRAN 77 original implementation in the `getrf`, `gbtrf`, and `potrf` interfaces.
- If `jobz` argument value specifies presence or absence of `z` argument, then `z` is always declared as optional and `jobz` is restored depending on whether `z` is present or not. It is not always so in the Netlib version (see "[Modified Netlib Interfaces](#)" in Appendix E).
- To avoid double error checking, processing of the *info* argument is limited to checking of the allocated memory and disarranging of optional arguments.
- If an argument that is present in the list of arguments completely defines another argument, the latter is always declared as optional.

You can transform an application that uses the Netlib LAPACK interfaces to ensure its work with the Intel MKL interfaces providing that:

- a. The application is correct, that is, unambiguous, compiler-independent, and contains no errors.
- b. Each routine name denotes only one specific routine. If any routine name in the application coincides with a name of the original Netlib routine (for example, after removing the `LA_` prefix) but denotes a routine different from the Netlib original routine, this name should be modified through context name replacement.

You should transform your application in the following cases (see Appendix E for specific differences of individual interfaces):

- When using the Netlib routines that differ from the Intel MKL routines only by the `LA_` prefix or in the array attribute `target`. The only transformation required in this case is context name replacement. See "[Interfaces Identical to Netlib](#)" in Appendix E for details.
- When using Netlib routines that differ from the Intel MKL routines by the `LA_` prefix, the `target` array attribute, and the names of formal arguments. In the case of positional passing of arguments, no additional transformation except context name replacement is required. In the case of the keywords passing of arguments, in addition to the context name replacement the names of mismatching keywords should also be modified. See "[Interfaces with Replaced Argument Names](#)" in Appendix E for details.
- When using the Netlib routines that differ from the respective Intel MKL routines by the `LA_` prefix, the `target` array attribute, sequence of the arguments, arguments missing in Intel MKL but present in Netlib and, vice versa, present in Intel MKL but missing in Netlib. Remove the differences in the sequence and range of the arguments in process of all the transformations when you use the Netlib routines specified by this bullet and the preceding bullet. See "[Modified Netlib Interfaces](#)" in Appendix E for details.

- When using the `getrf`, `gbtrf`, and `potrf` interfaces, that is, new functionality implemented in Intel MKL but unavailable in the Netlib source. To override the differences, build the desired functionality explicitly with the Intel MKL means or create a new subroutine with the new functionality, using specific MKL interfaces corresponding to LAPACK 77 routines. You can call the LAPACK 77 routines directly but using the new Intel MKL interfaces is preferable. See "[Interfaces Absent From Netlib](#)" and "[Interfaces of New Functionality](#)" in Appendix E for details. Note that if the transformed application calls `getrf`, `gbtrf` or `potrf` without controlling arguments `rcond` and `norm`, just context name replacement is enough in modifying the calls into the Intel MKL interfaces, as described in the first bullet above. The Netlib functionality is preserved in such cases.
- When using the Netlib auxiliary routines. In this case, call a corresponding subroutine directly, using the Intel MKL LAPACK 77 interfaces.

Transform your application as follows:

1. Make sure conditions a. and b. are met.
2. Select Netlib LAPACK 95 calls. For each call, do the following:
 - Select the type of digression and do the required transformations.
 - Revise results to eliminate unneeded code or data, which may appear after several identical calls.
3. Make sure the transformations are correct and complete.

Matrix Storage Schemes

LAPACK routines use the following matrix storage schemes:

- Full storage:** an m -by- n matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} ($i = 1..m$ $j = 1..n$), and stored in the array element $a(i, j)$.
- Packed storage** scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly: the upper or lower triangle of the matrix is packed by columns in a one-dimensional array.
- Band storage:** an m -by- n band matrix with k_l sub-diagonals and k_u superdiagonals is stored compactly in a two-dimensional array ab with k_l+k_u+1 rows and n columns. Columns of the matrix are stored in the corresponding columns of the array, and *diagonals* of the matrix are stored in rows of the array.
- Rectangular Full Packed (RFP) storage:** the upper or lower triangle of the matrix is packed combining the full and packed storage schemes. This combination enables using half of the full storage as packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels as the full storage.

Generally in LAPACK routines, arrays that hold matrices in packed storage have names ending in *p*; arrays with matrices in band storage have names ending in *b*; arrays with matrices in the RFP storage have names ending in *fp*.

For more information on matrix storage schemes, see "[Matrix Arguments](#)" in Appendix B.

Mathematical Notation

Descriptions of LAPACK routines use the following notation:

$Ax = b$	A system of linear equations with an n -by- n matrix $A = \{a_{ij}\}$, a right-hand side vector $b = \{b_i\}$, and an unknown vector $x = \{x_i\}$.
$AX = B$	A set of systems with a common matrix A and multiple right-hand sides. The columns of B are individual right-hand sides, and the columns of X are the corresponding solutions.
$ x $	the vector with elements $ x_i $ (absolute values of x_i).
$ A $	the matrix with elements $ a_{ij} $ (absolute values of a_{ij}).
$\ x\ _\infty = \max_i x_i $	The <i>infinity-norm</i> of the vector x .

$\ A\ _\infty = \max_i \sum_j a_{ij} $	The <i>infinity-norm</i> of the matrix A .
$\ A\ _1 = \max_j \sum_i a_{ij} $	The <i>one-norm</i> of the matrix A . $\ A\ _1 = \ A^T\ _\infty = \ A^H\ _\infty$
$\ x\ _2$	The <i>2-norm</i> of the vector x : $\ x\ _2 = (\sum_i x_i ^2)^{1/2} = \ x\ _E$.
$\ A\ _2$	The <i>2-norm (or spectral norm)</i> of the matrix A .
$\ A\ _2 = \max_i \sigma_i$, $\ A\ _2^2 = \max_{\ x\ =1} (Ax \cdot Ax)$.	
$\ A\ _E$	The <i>Euclidean norm</i> of the matrix A : $\ A\ _E^2 = \sum_i \sum_j a_{ij} ^2$ (for vectors, the Euclidean norm and the 2-norm are equal: $\ x\ _E = \ x\ _2$).
$\kappa(A) = \ A\ _1 \ A^{-1}\ _1$	The <i>condition number</i> of the matrix A .
λ_i	<i>Eigenvalues</i> of the matrix A (for the definition of eigenvalues, see Eigenvalue Problems).
σ_i	<i>Singular values</i> of the matrix A . They are equal to square roots of the eigenvalues of $A^H A$. (For more information, see Singular Value Decomposition).
$q(x, y)$	The <i>acute angle between vectors</i> x and y :
	$\cos q(x, y) = x \cdot y / (\ x\ _2 \ y\ _2)$.

Error Analysis

In practice, most computations are performed with rounding errors. Besides, you often need to solve a system $Ax = b$, where the data (the elements of A and b) are not known exactly. Therefore, it is important to understand how the data errors and rounding errors can affect the solution x .

Data perturbations. If x is the exact solution of $Ax = b$, and $x + \delta x$ is the exact solution of a perturbed problem $(A + \delta A)x = (b + \delta b)$, then

where

$$\kappa(A) = \|A\| \|A^{-1}\|.$$

In other words, relative errors in A or b may be amplified in the solution vector x by a factor $\kappa(A) = \|A\| \|A^{-1}\|$ called the *condition number* of A .

Rounding errors have the same effect as relative perturbations $c(n)\varepsilon$ in the original data. Here ε is the *machine precision*, and $c(n)$ is a modest function of the matrix order n . The corresponding solution error is $\|\delta x\| / \|x\| \leq c(n)\kappa(A)\varepsilon$. (The value of $c(n)$ is seldom greater than $10n$.)

Thus, if your matrix A is *ill-conditioned* (that is, its condition number $\kappa(A)$ is very large), then the error in the solution x is also large; you may even encounter a complete loss of precision. LAPACK provides routines that allow you to estimate $\kappa(A)$ (see [Routines for Estimating the Condition Number](#)) and also give you a more precise estimate for the actual solution error (see [Refining the Solution and Estimating Its Error](#)).

Linear Equation Routines

This section describes routines for performing the following computations:

- factoring the matrix (except for triangular matrices)
- equilibrating the matrix (except for RFP matrices)
- solving a system of linear equations
- estimating the condition number of a matrix (except for RFP matrices)
- refining the solution of linear equations and computing its error bounds (except for RFP matrices)
- inverting the matrix.

To solve a particular problem, you can call two or more [computational routines](#) or call a corresponding [driver routine](#) that combines several tasks in one call. For example, to solve a system of linear equations with a general matrix, call `?getrf` (*LU* factorization) and then `?getrs` (computing the solution). Then, call `?gerfs` to refine the solution and get the error bounds. Alternatively, use the driver routine `?gesvx` that performs all these tasks in one call.

Computational Routines

[Table "Computational Routines for Systems of Equations with Real Matrices"](#) lists the LAPACK computational routines (FORTRAN 77 and Fortran 95 interfaces) for factorizing, equilibrating, and inverting *real* matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error. [Table "Computational Routines for Systems of Equations with Complex Matrices"](#) lists similar routines for *complex* matrices. Respective routine names in the Fortran 95 interface are without the first symbol (see [Routine Naming Conventions](#)).

Computational Routines for Systems of Equations with Real Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	<code>?getrf</code>	<code>?geequ,</code> <code>?gequb</code>	<code>?getrs</code>	<code>?gecon</code>	<code>?gerfs,</code> <code>?gerfsx</code>	<code>?getri</code>
general band	<code>?gbtrf</code>	<code>?gbequ,</code> <code>?gbequb</code>	<code>?gbtrs</code>	<code>?gbcon</code>	<code>?gbrfs,</code> <code>?gbrfsx</code>	
general tridiagonal	<code>?gttrf</code>		<code>?gttrs</code>	<code>?gtcon</code>	<code>?gtrfs</code>	
diagonally dominant tridiagonal	<code>?dttrfb</code>		<code>?dttrsrb</code>			
symmetric positive-definite	<code>?potrf</code>	<code>?poequ,</code> <code>?poequb</code>	<code>?potrs</code>	<code>?pocon</code>	<code>?porfs,</code> <code>?porfsx</code>	<code>?potri</code>
symmetric positive-definite, packed storage	<code>?pptrf</code>	<code>?ppequ</code>	<code>?pptrs</code>	<code>?ppcon</code>	<code>?pprfs</code>	<code>?pptri</code>
symmetric positive-definite, RFP storage	<code>?pftrf</code>		<code>?pftrs</code>			<code>?pftri</code>
symmetric positive-definite, band	<code>?pbtrf</code>	<code>?pbequ</code>	<code>?pbtrs</code>	<code>?pbcon</code>	<code>?pbrfs</code>	

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
symmetric positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
symmetric indefinite	?sytrf ?sytrf_rook	?syequb	?sytrs ?sytrs_rook ?sytrs2	?sycon ?sycon_rook ?sycon_rook	?syrfss, ?syrfssx	?sytri ?sytri_rook ?sytri2 ?sytri2x
symmetric indefinite, packed storage	?sptrf		?sptrs	?spcon	?sprfs	?sptri
triangular			?trtrs	?trcon	?trrfs	?trtri
triangular, packed storage			?tptrs	?tpcon	?tpdfs	?tptri
triangular, RFP storage						?tftri
triangular band			?tbtrs	?tbcon	?tbrfs	

In the table above, ? denotes s (single precision) or d (double precision) for the FORTRAN 77 interface.

Computational Routines for Systems of Equations with Complex Matrices

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general	?getrf	?geequ, ?geequb	?getrs	?gecon	?gerfs, ?gerfsx	?getri
general band	?gbtrf	?gbequ, ?gbequb	?gbtrs	?gbcon	?gbrfs, ?gbrfsx	
general tridiagonal	?gttrf		?gttrs	?gtcon	?gtrfs	
Hermitian positive-definite	?potrf	?poequ, ?poequb	?potrs	?pocon	?porfs, ?porfsx	?potri
Hermitian positive-definite, packed storage	?pptrf	?ppequ	?pptrs	?ppcon	?pprfs	?pptri
Hermitian positive-definite, RFP storage	?pftrf		?pftrs			?pftri
Hermitian positive-definite, band	?pbtrf	?pbequ	?pbtrs	?pbcon	?pbrfs	
Hermitian positive-definite, tridiagonal	?pttrf		?pttrs	?ptcon	?ptrfs	
Hermitian indefinite	?hetrf	?heequb	?hetrs	?hecon	?herfs, ?herfsx	?hetri

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
	?hetrf_rook		?hetrs_rook?hecon_rook ?herfsx ?hetrs2		?hetri_rook ?hetri2 ?hetri2x	
symmetric indefinite	?sytrf ?syequb ?sytrf_rook		?sytrs ?sycon ?sytrs_rook?syconv ?sytrs2 ?sycon_rook		?syrfs, ?sytri ?syrfsx ?sytri_rook ?sytrii ?sytrii2 ?sytrii2x	
Hermitian indefinite, packed storage	?hptra		?hptrs	?hpcon	?hprrfs	?hptri
symmetric indefinite, packed storage	?sptra		?sptrs	?spcon	?sprfs	?sptri
triangular			?trtrs	?trcon	?trrrfs	?trtri
triangular, packed storage			?tptrs	?tpcon	?tprrfs	?tptri
triangular, RFP storage						?tftri
triangular band			?tbtrs	?tbcon	?tbrfs	

In the table above, ? stands for c (single precision complex) or z (double precision complex) for FORTRAN 77 interface.

Routines for Matrix Factorization

This section describes the LAPACK routines for matrix factorization. The following factorizations are supported:

- LU factorization
- Cholesky factorization of real symmetric positive-definite matrices
- Cholesky factorization of real symmetric positive-definite matrices with pivoting
- Cholesky factorization of Hermitian positive-definite matrices
- Cholesky factorization of Hermitian positive-definite matrices with pivoting
- Bunch-Kaufman factorization of real and complex symmetric matrices
- Bunch-Kaufman factorization of Hermitian matrices.

You can compute:

- the LU factorization using full and band storage of matrices
- the Cholesky factorization using full, packed, RFP, and band storage
- the Bunch-Kaufman factorization using full and packed storage.

?getrf

Computes the LU factorization of a general m-by-n matrix.

Syntax

FORTRAN 77:

```
call sgetrf( m, n, a, lda, ipiv, info )
call dgetrf( m, n, a, lda, ipiv, info )
call cgetrf( m, n, a, lda, ipiv, info )
call zgetrf( m, n, a, lda, ipiv, info )
```

Fortran 95:

```
call getrf( a [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>getrf( int matrix_layout, lapack_int m, lapack_int n, <datatype>* a, lapack_int lda, lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the *LU* factorization of a general m -by- n matrix A as

$$A = P \cdot L \cdot U,$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in A ; $n \geq 0$.
<i>a</i>	REAL for sgetrf DOUBLE PRECISION for dgetrf COMPLEX for cgetrf DOUBLE COMPLEX for zgetrf. Array, DIMENSION (<i>lda</i> , *). Contains the matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of array <i>a</i> .

Output Parameters

a Overwritten by *L* and *U*. The unit diagonal elements of *L* are not stored.

ipiv INTEGER.

Array, DIMENSION at least $\max(1, \min(m, n))$. The pivot indices; for $1 \leq i \leq \min(m, n)$, row *i* was interchanged with row *ipiv(i)*.

info INTEGER. If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `getrf` interface are as follows:

a Holds the matrix *A* of size (*m*, *n*).

ipiv Holds the vector of length $\min(m, n)$.

Application Notes

The computed *L* and *U* are the exact factors of a perturbed matrix *A* + *E*, where

$$\|E\| \leq c(\min(m, n))\epsilon \|A\| \|U\|$$

c(n) is a modest linear function of *n*, and ϵ is the machine precision.

The approximate number of floating-point operations for real flavors is

$$(2/3)n^3 \quad \text{If } m = n,$$

$$(1/3)n^2(3m-n) \quad \text{If } m > n,$$

$$(1/3)m^2(3n-m) \quad \text{If } m < n.$$

The number of operations for complex flavors is four times greater.

After calling this routine with *m* = *n*, you can call the following:

?getrs to solve $A^*x = B$ or $A^T x = B$ or $A^H x = B$

?gecon to estimate the condition number of *A*

?getri to compute the inverse of *A*.

See Also

[mkl_progress](#)

?gbtrf

Computes the LU factorization of a general m-by-n band matrix.

Syntax**FORTRAN 77:**

```
call sgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbtrf( m, n, kl, ku, ab, ldab, ipiv, info )
```

Fortran 95:

```
call gbtrf( ab [,kl] [,m] [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>gbtrf( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, <datatype>* ab, lapack_int ldab, lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the *LU* factorization of a general m -by- n band matrix A with kl non-zero subdiagonals and ku non-zero superdiagonals, that is,

$$A = P \cdot L \cdot U,$$

where P is a permutation matrix; L is lower triangular with unit diagonal elements and at most kl non-zero elements in each column; U is an upper triangular band matrix with $kl + ku$ superdiagonals. The routine uses partial pivoting, with row interchanges (which creates the additional kl superdiagonals in U).

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows in matrix A ; $m \geq 0$.
<i>n</i>	INTEGER. The number of columns in matrix A ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
<i>ab</i>	REAL for sgbtrf DOUBLE PRECISION for dgbtrf

COMPLEX for `cgbtrf`

DOUBLE COMPLEX for `zgbtrf`.

Array, DIMENSION ($l\text{dab}, *$). The array ab contains the matrix A in band storage, in rows $k_1 + 1$ to $2*k_1 + k_u + 1$; rows 1 to k_1 of the array need not be set. The j -th column of A is stored in the j -th column of the array ab as follows:

$ab(k_1 + k_u + 1 + i - j, j) = a(i, j)$ for $\max(1, j-k_u) \leq i \leq \min(m, j+k_1)$.

$l\text{dab}$

INTEGER. The leading dimension of the array ab . ($l\text{dab} \geq 2*k_1 + k_u + 1$)

Output Parameters

ab

Overwritten by L and U . U is stored as an upper triangular band matrix with $k_1 + k_u$ superdiagonals in rows 1 to $k_1 + k_u + 1$, and the multipliers used during the factorization are stored in rows $k_1 + k_u + 2$ to $2*k_1 + k_u + 1$. See Application Notes below for further details.

$ipiv$

INTEGER.
Array, DIMENSION at least $\max(1, \min(m, n))$. The pivot indices; for $1 \leq i \leq \min(m, n)$, row i was interchanged with row $ipiv(i)$.

$info$

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbtrf` interface are as follows:

ab

Holds the array A of size $(2*k_1+k_u+1, n)$.

$ipiv$

Holds the vector of length $\min(m, n)$.

k_1

If omitted, assumed $k_1 = k_u$.

k_u

Restored as $k_u = l\text{da}-2*k_1-1$.

m

If omitted, assumed $m = n$.

Application Notes

The computed L and U are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(k_1+k_u+1) \cdot \epsilon \cdot P |L| |U|$$

$c(k)$ is a modest linear function of k , and ϵ is the machine precision.

The total number of floating-point operations for real flavors varies between approximately $2n(ku+1)k_1$ and $2n(k_1+ku+1)k_1$. The number of operations for complex flavors is four times greater. All these estimates assume that k_1 and ku are much less than $\min(m, n)$.

The band storage scheme is illustrated by the following example, when $m = n = 6$, $k_1 = 2$, $ku = 1$:

on entry							on exit						
$\begin{bmatrix} * & * & * & + & + & + \\ * & * & + & + & + & + \\ * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{bmatrix}$							$\begin{bmatrix} * & * & * & u_{14} & u_{25} & u_{36} \\ * & * & u_{13} & u_{24} & u_{35} & u_{46} \\ * & u_{12} & u_{23} & u_{34} & u_{45} & u_{56} \\ u_{11} & u_{22} & u_{33} & u_{44} & u_{55} & u_{66} \\ m_{21} & m_{32} & m_{43} & m_{54} & m_{65} & * \\ m_{31} & m_{42} & m_{53} & m_{64} & * & * \end{bmatrix}$						

Array elements marked * are not used by the routine; elements marked + need not be set on entry, but are required by the routine to store elements of U because of fill-in resulting from the row interchanges.

After calling this routine with $m = n$, you can call the following routines:

[gbtrs](#) to solve $A^*X = B$ or $A^T*X = B$ or $A^H*X = B$

[gbcon](#) to estimate the condition number of A .

See Also

[mkl_progress](#)

?gttrf

Computes the LU factorization of a tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sgttrf( n, dl, d, du, du2, ipiv, info )
call dgttrf( n, dl, d, du, du2, ipiv, info )
call cgttrf( n, dl, d, du, du2, ipiv, info )
call zgttrf( n, dl, d, du, du2, ipiv, info )
```

Fortran 95:

```
call gttrf( dl, d, du, du2 [, ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>gttrf( lapack_int n, <datatype>* dl, <datatype>* d, <datatype>* du, <datatype>* du2, lapack_int* ipiv );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`

- C: mkl.h

Description

The routine computes the *LU* factorization of a real or complex tridiagonal matrix A using elimination with partial pivoting and row interchanges.

The factorization has the form

$$A = L^*U,$$

where L is a product of permutation and unit lower bidiagonal matrices and U is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>d1, d, du</i>	REAL for sgttrf DOUBLE PRECISION for dgttrf COMPLEX for cgttrf DOUBLE COMPLEX for zgttrf. Arrays containing elements of A . The array <i>d1</i> of dimension $(n - 1)$ contains the subdiagonal elements of A . The array <i>d</i> of dimension n contains the diagonal elements of A . The array <i>du</i> of dimension $(n - 1)$ contains the superdiagonal elements of A .
<i>dl</i>	Overwritten by the $(n-1)$ multipliers that define the matrix L from the <i>LU</i> factorization of A . The matrix L has unit diagonal elements, and the $(n-1)$ elements of <i>dl</i> form the subdiagonal. All other elements of L are zero.
<i>d</i>	Overwritten by the n diagonal elements of the upper triangular matrix U from the <i>LU</i> factorization of A .
<i>du</i>	Overwritten by the $(n-1)$ elements of the first superdiagonal of U .
<i>du2</i>	REAL for sgttrf DOUBLE PRECISION for dgttrf COMPLEX for cgttrf DOUBLE COMPLEX for zgttrf. Array, dimension $(n - 2)$. On exit, <i>du2</i> contains $(n-2)$ elements of the second superdiagonal of U .
<i>ipiv</i>	INTEGER.

Output Parameters

<i>dl</i>	Overwritten by the $(n-1)$ multipliers that define the matrix L from the <i>LU</i> factorization of A . The matrix L has unit diagonal elements, and the $(n-1)$ elements of <i>dl</i> form the subdiagonal. All other elements of L are zero.
<i>d</i>	Overwritten by the n diagonal elements of the upper triangular matrix U from the <i>LU</i> factorization of A .
<i>du</i>	Overwritten by the $(n-1)$ elements of the first superdiagonal of U .
<i>du2</i>	REAL for sgttrf DOUBLE PRECISION for dgttrf COMPLEX for cgttrf DOUBLE COMPLEX for zgttrf. Array, dimension $(n - 2)$. On exit, <i>du2</i> contains $(n-2)$ elements of the second superdiagonal of U .
<i>ipiv</i>	INTEGER.

Array, dimension (n). The pivot indices: for $1 \leq i \leq n$, row i was interchanged with row $ipiv(i)$. $ipiv(i)$ is always i or $i+1$; $ipiv(i) = i$ indicates a row interchange was not required.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by zero will occur if you use the factor U for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gttrf` interface are as follows:

<i>dl</i>	Holds the vector of length ($n-1$).
<i>d</i>	Holds the vector of length n .
<i>du</i>	Holds the vector of length ($n-1$).
<i>du2</i>	Holds the vector of length ($n-2$).
<i>ipiv</i>	Holds the vector of length n .

Application Notes

?gbtrs	to solve $A^*X = B$ or $A^T*X = B$ or $A^H*X = B$
?gbcon	to estimate the condition number of A .

?dttrfb

Computes the factorization of a diagonally dominant tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sdttrfb( n, dl, d, du, info )
call ddttrfb( n, dl, d, du, info )
call cdttrfb( n, dl, d, du, info )
call zdttrfb( n, dl, d, du, info )
```

Fortran 95:

```
call dttrfb( dl, d, du [, info] )
```

C:

```
void sdttrfb (const MKL_INT * n , float * dl , float * d , const float * du , MKL_INT * info );
```

```

void ddttrfb_ (const MKL_INT * n , double * dl , double * d , const double * du ,
MKL_INT * info );

void cdttrfb_ (const MKL_INT * n , MKL_Complex8 * dl , MKL_Complex8 * d , const
MKL_Complex8 * du , MKL_INT * info );

void zdttrfb_ (const MKL_INT * n , MKL_Complex16 * dl , MKL_Complex16 * d , const
MKL_Complex16 * du , MKL_INT * info );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The ?dttrfb routine computes the factorization of a real or complex tridiagonal matrix A with the BABE (Burning At Both Ends) algorithm without pivoting. The factorization has the form

$$A = L_1 * U * L_2$$

where

- L_1 and L_2 are unit lower bidiagonal with k and $n - k - 1$ subdiagonal elements, respectively, where $k = n/2$, and
- U is an upper bidiagonal matrix with nonzeros in only the main diagonal and first superdiagonal.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>dl, d, du</i>	REAL for sdttrfb DOUBLE PRECISION for ddttrfb COMPLEX for cdttrfb DOUBLE COMPLEX for zdttrfb. Arrays containing elements of A . The array <i>dl</i> of dimension $(n - 1)$ contains the subdiagonal elements of A . The array <i>d</i> of dimension <i>n</i> contains the diagonal elements of A . The array <i>du</i> of dimension $(n - 1)$ contains the superdiagonal elements of A .
<i>info</i>	None

Output Parameters

<i>dl</i>	Overwritten by the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A .
<i>d</i>	Overwritten by the <i>n</i> diagonal element reciprocals of the upper triangular matrix U from the factorization of A .
<i>du</i>	Overwritten by the $(n-1)$ elements of the superdiagonal of U .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by zero will occur if you use the factor U for solving a system of linear equations.

Application Notes

A diagonally dominant tridiagonal system is defined such that $|d_i| > |dl_{i-1}| + |du_i|$ for any i :

$1 < i < n$, and $|d_1| > |du_1|$, $|d_n| > |dl_{n-1}|$

The underlying BABE algorithm is designed for diagonally dominant systems. Such systems are free from the numerical stability issue unlike the canonical systems that use elimination with partial pivoting (see [?gttrf](#)). The diagonally dominant systems are much faster than the canonical systems.

NOTE

- The current implementation of BABE has a potential accuracy issue on very small or large data close to the underflow or overflow threshold respectively. Scale the matrix before applying the solver in the case of such input data.
 - Applying the [?dttrfb](#) factorization to non-diagonally dominant systems may lead to an accuracy loss, or false singularity detected due to no pivoting.
-

?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```
call spotrf( uplo, n, a, lda, info )
call dpotrf( uplo, n, a, lda, info )
call cpotrf( uplo, n, a, lda, info )
call zpotrf( uplo, n, a, lda, info )
```

Fortran 95:

```
call potrf( a [, uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>potrf( int matrix_layout, char uplo, lapack_int n, <datatype>* a,
lapack_int lda );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A :

$A = U^T * U$ for real data, $A = U^H * U$ for complex data if $uplo='U'$
 $A = L * L^T$ for real data, $A = L * L^H$ for complex data if $uplo='L'$
 where L is a lower triangular matrix and U is upper triangular.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <i>uplo</i> = ' <i>U</i> ', the array <i>a</i> stores the upper triangular part of the matrix A . If <i>uplo</i> = ' <i>L</i> ', the array <i>a</i> stores the lower triangular part of the matrix A .
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a</i>	REAL for <i>spotrf</i> DOUBLE PRECISION for <i>dpotrf</i> COMPLEX for <i>cpotrf</i> DOUBLE COMPLEX for <i>zpotrf</i> . Array, DIMENSION (<i>lda</i> , *). The array <i>a</i> contains either the upper or the lower triangular part of the matrix A (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> .

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by the Cholesky factor U or L , as specified by <i>uplo</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *potrf* interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

If $uplo = \text{'U'}$, the computed factor U is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\varepsilon |U^H| |U|, |e_{ij}| \leq c(n)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for $uplo = \text{'L'}$.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

?potrs	to solve $A^*X = B$
?pocon	to estimate the condition number of A
?potri	to compute the inverse of A .

See Also

[mkl_progress](#)

?pstrf

Computes the Cholesky factorization with complete pivoting of a real symmetric (complex Hermitian) positive semidefinite matrix.

Syntax

FORTRAN 77:

```
call spstrf( uplo, n, a, lda, piv, rank, tol, work, info )
call dpstrf( uplo, n, a, lda, piv, rank, tol, work, info )
call cpstrf( uplo, n, a, lda, piv, rank, tol, work, info )
call zpstrf( uplo, n, a, lda, piv, rank, tol, work, info )
```

C:

```
lapack_int LAPACKE_spstrf( int matrix_layout, char uplo, lapack_int n, float* a,
                           lapack_int lda, lapack_int* piv, lapack_int* rank, float tol );
lapack_int LAPACKE_dpstrf( int matrix_layout, char uplo, lapack_int n, double* a,
                           lapack_int lda, lapack_int* piv, lapack_int* rank, double tol );
lapack_int LAPACKE_cpstrf( int matrix_layout, char uplo, lapack_int n,
                           lapack_complex_float* a, lapack_int lda, lapack_int* piv, lapack_int* rank, float tol );
lapack_int LAPACKE_zpstrf( int matrix_layout, char uplo, lapack_int n,
                           lapack_complex_double* a, lapack_int lda, lapack_int* piv, lapack_int* rank, double tol );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the Cholesky factorization with complete pivoting of a real symmetric (complex Hermitian) positive semidefinite matrix. The form of the factorization is:

$$\begin{aligned} P^T * A * P &= U^T * U, \text{ if } \text{uplo} = 'U' \text{ for real flavors,} \\ P^H * A * P &= U^H * U, \text{ if } \text{uplo} = 'U' \text{ for complex flavors,} \\ P^T * A * P &= L * L^T, \text{ if } \text{uplo} = 'L' \text{ for real flavors,} \\ P^H * A * P &= L * L^H, \text{ if } \text{uplo} = 'L' \text{ for complex flavors,} \end{aligned}$$

where P is stored as vector piv , ' U ' and ' L ' are upper and lower triangular matrices respectively.

This algorithm does not attempt to check that A is positive semidefinite. This version of the algorithm calls level 3 BLAS.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be ' U ' or ' L '.
	Indicates whether the upper or lower triangular part of A is stored: If $uplo = 'U'$, the array a stores the upper triangular part of the matrix A , and the strictly lower triangular part of the matrix is not referenced. If $uplo = 'L'$, the array a stores the lower triangular part of the matrix A , and the strictly upper triangular part of the matrix is not referenced.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a, work</i>	REAL for spstrf DOUBLE PRECISION for dpstrf COMPLEX for cpstrf DOUBLE COMPLEX for zpstrf. Array a , DIMENSION ($lda, *$). The array a contains either the upper or the lower triangular part of the matrix A (see <i>uplo</i>). The second dimension of a must be at least $\max(1, n)$. $work(*)$ is a workspace array. The dimension of $work$ is at least $\max(1, 2*n)$.
<i>tol</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

User defined tolerance. If $tol < 0$, then $n*U*\max(a(k,k))$ will be used. The algorithm terminates at the $(k-1)$ -th step, if the pivot $\leq tol$.

*lda*INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

Output Parameters

*a*If $info = 0$, the factor *U* or *L* from the Cholesky factorization is as described in *Description*.*piv*

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array *piv* is such that the nonzero entries are $p(piv(k), k) = 1$.*rank*

INTEGER.

The rank of *a* given by the number of steps the algorithm completed.*info*INTEGER. If $info = 0$, the execution is successful.If $info = -k$, the k -th argument had an illegal value.If $info > 0$, the matrix *A* is either rank deficient with a computed rank as returned in *rank*, or is indefinite.

?pftrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using the Rectangular Full Packed (RFP) format .

Syntax

FORTRAN 77:

```
call spftrf( transr, uplo, n, a, info )
call dpftrf( transr, uplo, n, a, info )
call cpftrf( transr, uplo, n, a, info )
call zpftrf( transr, uplo, n, a, info )
```

C:

```
lapack_int LAPACKE_<?>pftrf( int matrix_layout, char transr, char uplo, lapack_int n,
<datatype>* a );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, a Hermitian positive-definite matrix *A*:

$A = U^T * U$ for real data, $A = U^H * U$ for complex data	if $uplo='U'$
$A = L * L^T$ for real data, $A = L * L^H$ for complex data	if $uplo='L'$

where L is a lower triangular matrix and U is upper triangular.

The matrix A is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

This is the block version of the algorithm, calling Level 3 BLAS.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>transr</i>	CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data). If <i>transr</i> = 'N', the Normal <i>transr</i> of RFP A is stored. If <i>transr</i> = 'T', the Transpose <i>transr</i> of RFP A is stored. If <i>transr</i> = 'C', the Conjugate-Transpose <i>transr</i> of RFP A is stored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A . If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i>	REAL for <i>spftrf</i> DOUBLE PRECISION for <i>dpftrf</i> COMPLEX for <i>cpftrf</i> DOUBLE COMPLEX for <i>zpftrf</i> . Array, DIMENSION ($n * (n+1) / 2$). The array <i>a</i> contains the matrix A in the RFP format.

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by the Cholesky factor U or L , as specified by <i>info</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value. If <i>info</i> = i , the leading minor of order <i>i</i> (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A .

?pptrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using packed storage.

Syntax**FORTRAN 77:**

```
call spptrf( uplo, n, ap, info )
call dpptrf( uplo, n, ap, info )
call cpptrf( uplo, n, ap, info )
call zpptrf( uplo, n, ap, info )
```

Fortran 95:

```
call pptrf( ap [, uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>pptrf( int matrix_layout, char uplo, lapack_int n, <datatype>* ap );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite packed matrix A :

$$\begin{aligned} A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo='U' \\ A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo='L' \end{aligned}$$

where L is a lower triangular matrix and U is upper triangular.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

`uplo`

CHARACTER*1. Must be '`U`' or '`L`'.

Indicates whether the upper or lower triangular part of A is packed in the array `ap`, and how A is factored:

If `uplo = 'U'`, the array `ap` stores the upper triangular part of the matrix A , and A is factored as $U^H * U$.

If `uplo = 'L'`, the array `ap` stores the lower triangular part of the matrix A ; A is factored as $L * L^H$.

n INTEGER. The order of matrix A ; $n \geq 0$.

ap REAL for `spptrf`
DOUBLE PRECISION for `dpptrf`
COMPLEX for `cpptrf`
DOUBLE COMPLEX for `zpptrf`.
Array, DIMENSION at least $\max(1, n(n+1)/2)$. The array *ap* contains either the upper or the lower triangular part of the matrix A (as specified by *uplo*) in packed storage (see [Matrix Storage Schemes](#)).

Output Parameters

ap The upper or lower triangular part of A in packed storage is overwritten by the Cholesky factor U or L , as specified by *uplo*.

info INTEGER. If *info*=0, the execution is successful.
If *info* = $-i$, the *i*-th parameter had an illegal value.
If *info* = i , the leading minor of order *i* (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pptrf` interface are as follows:

<i>ap</i>	Holds the array A of size $(n^*(n+1)/2)$.
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

If *uplo* = '`U`', the computed factor U is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\varepsilon |U^H| |U|, |e_{ij}| \leq c(n)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for *uplo* = '`L`'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

? <code>pptrs</code>	to solve $A^*X = B$
? <code>ppcon</code>	to estimate the condition number of A

See Also

mkl progress

?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite band matrix.

Syntax

FORTRAN 77:

```
call spbtrf( uplo, n, kd, ab, ldab, info )
call dpbtrf( uplo, n, kd, ab, ldab, info )
call cpbtrf( uplo, n, kd, ab, ldab, info )
call zpbtrf( uplo, n, kd, ab, ldab, info )
```

Fortran 95:

```
call pbtrf( ab [, uplo] [, info] )
```

C_i

```
lapack_int LAPACKE_<?>pbtrf( int matrix_layout, char uplo, lapack_int n, lapack_int kd,  
<datatype>* ab, lapack_int ldab );
```

Include Files

- Fortran: mkl.fi
 - Fortran 95: lapack.f90
 - C: mkl.h

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite band matrix A :

$A = U^T * U$ for real data, $A = U^H * U$ for complex data if $\text{uplo} = 'U'$
 $A = L * L^T$ for real data, $A = L * L^H$ for complex data if $\text{uplo} = 'L'$

where L is a lower triangular matrix and U is upper triangular.

NOTE

NOTE This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored in the array ab , and how A is factored:

	If $uplo = 'U'$, the upper triangle of A is stored.
	If $uplo = 'L'$, the lower triangle of A is stored.
n	INTEGER. The order of matrix A ; $n \geq 0$.
kd	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
ab	REAL for <code>s pbtrf</code> DOUBLE PRECISION for <code>d pbtrf</code> COMPLEX for <code>c pbtrf</code> DOUBLE COMPLEX for <code>z pbtrf</code> . Array, DIMENSION $(*, *)$. The array ab contains either the upper or the lower triangular part of the matrix A (as specified by $uplo$) in band storage (see Matrix Storage Schemes). The second dimension of ab must be at least $\max(1, n)$.
$ldab$	INTEGER. The leading dimension of the array ab . ($ldab \geq kd + 1$)

Output Parameters

ab	The upper or lower triangular part of A (in band storage) is overwritten by the Cholesky factor U or L , as specified by $uplo$.
$info$	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbtrf` interface are as follows:

ab	Holds the array A of size $(kd+1, n)$.
$uplo$	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

If $uplo = 'U'$, the computed factor U is the exact factor of a perturbed matrix $A + E$, where

$$|E| \leq c(kd + 1)\varepsilon |U^H| |U|, |e_{ij}| \leq c(kd + 1)\varepsilon \sqrt{a_{ii}a_{jj}}$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for $uplo = 'L'$.

The total number of floating-point operations for real flavors is approximately $n(kd+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that kd is much less than n .

After calling this routine, you can call the following routines:

?pbtrs	to solve $A^*X = B$
?pbcon	to estimate the condition number of A .

See Also

[mkl_progress](#)

?pttrf

Computes the factorization of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

FORTRAN 77:

```
call spttrf( n, d, e, info )
call dpttrf( n, d, e, info )
call cpttrf( n, d, e, info )
call zpttrf( n, d, e, info )
```

Fortran 95:

```
call pttrf( d, e [,info] )
```

C:

```
lapack_int LAPACKE_spttrf( lapack_int n, float* d, float* e );
lapack_int LAPACKE_dpttrf( lapack_int n, double* d, double* e );
lapack_int LAPACKE_cpttrf( lapack_int n, float* d, lapack_complex_float* e );
lapack_int LAPACKE_zpttrf( lapack_int n, double* d, lapack_complex_double* e );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite tridiagonal matrix A :

$A = L^*D*L^T$ for real flavors, or

$A = L^*D*L^H$ for complex flavors,

where D is diagonal and L is unit lower bidiagonal. The factorization may also be regarded as having the form $A = U^T*D*U$ for real flavors, or $A = U^H*D*U$ for complex flavors, where U is unit upper bidiagonal.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>d</i>	REAL for <code>spttrf</code> , <code>cpttrf</code> DOUBLE PRECISION for <code>dpttrf</code> , <code>zpttrf</code> . Array, dimension (n). Contains the diagonal elements of A .
<i>e</i>	REAL for <code>spttrf</code> DOUBLE PRECISION for <code>dpttrf</code> COMPLEX for <code>cpttrf</code> DOUBLE COMPLEX for <code>zpttrf</code> . Array, dimension ($n - 1$). Contains the subdiagonal elements of A .

Output Parameters

<i>d</i>	Overwritten by the n diagonal elements of the diagonal matrix D from the L^*D*L^T (for real flavors) or L^*D*L^H (for complex flavors) factorization of A .
<i>e</i>	Overwritten by the $(n - 1)$ off-diagonal elements of the unit bidiagonal factor L or U from the factorization of A .
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite; if $i < n$, the factorization could not be completed, while if $i = n$, the factorization was completed, but $d(n) \leq 0$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pttrf` interface are as follows:

<i>d</i>	Holds the vector of length n .
<i>e</i>	Holds the vector of length $(n-1)$.

?sytrf

Computes the Bunch-Kaufman factorization of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call csytrf( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytrf( uplo, n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call sytrf( a [, uplo] [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>sytrf( int matrix_layout, char uplo, lapack_int n, <datatype>* a,
lapack_int lda, lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the factorization of a real/complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

```
if uplo='U',  $A = P \cdot U \cdot D \cdot U^T \cdot P^T$   
if uplo='L',  $A = P \cdot L \cdot D \cdot L^T \cdot P^T$ ,
```

where A is the input matrix, P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE

This routine supports the Progress Routine feature. See [Progress Routine](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored and how A is factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A , and A is factored as $P \cdot U \cdot D \cdot U^T \cdot P^T$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A , and A is factored as $P \cdot L \cdot D \cdot L^T \cdot P^T$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a</i>	REAL for ssytrf DOUBLE PRECISION for dsytrf COMPLEX for csytrf DOUBLE COMPLEX for zsytrf. Array, DIMENSION (<i>lda</i> , *). The array <i>a</i> contains either the upper or the lower triangular part of the matrix A (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

work Same type as *a*. A workspace array, dimension at least $\max(1, lwork)$.

lwork INTEGER. The size of the *work* array (*lwork* $\geq n$).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by `xerbla`.

See [Application Notes](#) for the suggested value of *lwork*.

Output Parameters

a The upper or lower triangular part of *a* is overwritten by details of the block-diagonal matrix *D* and the multipliers used to obtain the factor *U* (or *L*).

work(1) If *info*=0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

ipiv INTEGER.

Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of *D*. If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

info INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, d_{ii} is 0. The factorization has been completed, but *D* is exactly singular. Division by 0 will occur if you use *D* for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytrf` interface are as follows:

a holds the matrix *A* of size (*n*, *n*)

ipiv holds the vector of length *n*

*uplo*must be '`U`' or '`L`'. The default value is '`U`'.

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the corresponding columns of the array *a*, but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv(i) = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array *a*.

If $uplo = 'U'$, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\varepsilon |P| |U| |D| |U^T| |P^T|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. A similar estimate holds for the computed L and D when $uplo = 'L'$.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

?sytrs	to solve $A^*X = B$
?sycon	to estimate the condition number of A
?sytri	to compute the inverse of A .

If $uplo = 'U'$, then $A = U^*D^*U'$, where

$$U = P(n)*U(n)* \dots *P(k)*U(k)* \dots ,$$

that is, U is a product of terms $P(k)*U(k)$, where

- k decreases from n to 1 in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by $ipiv(k)$.
- $U(k)$ is a unit upper triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$U(k) = \begin{pmatrix} I & v & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix} \begin{matrix} k-s \\ s \\ n-k \end{matrix}$$

$$\begin{matrix} k-s & s & n-k \end{matrix}$$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(1:k-1,k)$.

If $s = 2$, the upper triangle of $D(k)$ overwrites $A(k-1,k-1)$, $A(k-1,k)$ and $A(k,k)$, and v overwrites $A(1:k-2,k-1:k)$.

If $uplo = 'L'$, then $A = L^*D^*L'$, where

$L = P(1)^*L(1)^* \dots ^*P(k)^*L(k)^* \dots$

that is, L is a product of terms $P(k)^*L(k)$, where

- k increases from 1 to n in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by $ipiv(k)$.
- $L(k)$ is a unit lower triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$L(k) = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & v & I \end{pmatrix} \begin{matrix} k-1 \\ s \\ n-k-s+1 \end{matrix}$$

$$\begin{matrix} k-1 & s & n-k-s+1 \end{matrix}$$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(k+1:n,k)$.

If $s = 2$, the lower triangle of $D(k)$ overwrites $A(k,k)$, $A(k+1,k)$, and $A(k+1,k+1)$, and v overwrites $A(k+2:n,k:k+1)$.

See Also

[mkl_progress](#)

?sytrf_rook

Computes the bounded Bunch-Kaufman factorization of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call csytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call sytrf_rook( a [, uplo] [,ipiv] [,info] )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the factorization of a real/complex symmetric matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method. The form of the factorization is:

```
if uplo='U',  $A = U*D*U^T$ 
if uplo='L',  $A = L*D*L^T$ ,
```

where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored and how A is factored:
	If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A , and A is factored as $U*D*U^T$.
	If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A , and A is factored as $L*D*L^T$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a</i>	REAL for ssytrf_rook DOUBLE PRECISION for dsytrf_rook COMPLEX for csytrf_rook DOUBLE COMPLEX for zsytrf_rook. Array, size (<i>lda</i> , *). The array <i>a</i> contains either the upper or the lower triangular part of the matrix A (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>work</i>	Same type as <i>a</i> . A workspace array, dimension at least $\max(1, lwork)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array (<i>lwork</i> $\geq n$). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See ?sytrf Application Notes for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by details of the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>).
<i>work(1)</i>	If <i>info</i> =0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i> . If <i>ipiv(k) > 0</i> , then rows and columns <i>k</i> and <i>ipiv(k)</i> were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block. If <i>uplo = 'U'</i> and <i>ipiv(k) < 0</i> and <i>ipiv(k - 1) < 0</i> , then rows and columns <i>k</i> and $-ipiv(k)$ were interchanged, rows and columns <i>k - 1</i> and $-ipiv(k - 1)$ were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block. If <i>uplo = 'L'</i> and <i>ipiv(k) < 0</i> and <i>ipiv(k + 1) < 0</i> , then rows and columns <i>k</i> and $-ipiv(k)$ were interchanged, rows and columns <i>k + 1</i> and $-ipiv(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , D_{ii} is 0. The factorization has been completed, but <i>D</i> is exactly singular. Division by 0 will occur if you use <i>D</i> for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytrf_rook` interface are as follows:

<i>a</i>	holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>)
<i>ipiv</i>	holds the vector of length <i>n</i>
<i>uplo</i>	must be ' <i>U</i> ' or ' <i>L</i> '. The default value is ' <i>U</i> '.

Application Notes

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

<code>?sytrs_rook</code>	to solve $A^*X = B$
<code>?sycon_rook</code>	to estimate the condition number of <i>A</i>

[?sytri_rook](#)to compute the inverse of A .

If $uplo = 'U'$, then $A = U^*D^*U'$, where

$$U = P(n)*U(n)* \dots *P(k)*U(k)* \dots ,$$

that is, U is a product of terms $P(k)*U(k)$, where

- k decreases from n to 1 in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by $ipiv(k)$.
- $U(k)$ is a unit upper triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$U(k) = \begin{pmatrix} I & v & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{pmatrix}_{\begin{array}{c} k-s \\ s \\ n-k \end{array}}$$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(1:k-1,k)$.

If $s = 2$, the upper triangle of $D(k)$ overwrites $A(k-1,k-1)$, $A(k-1,k)$ and $A(k,k)$, and v overwrites $A(1:k-2,k-1:k)$.

If $uplo = 'L'$, then $A = L^*D^*L'$, where

$$L = P(1)*L(1)* \dots *P(k)*L(k)* \dots ,$$

that is, L is a product of terms $P(k)*L(k)$, where

- k increases from 1 to n in steps of 1 and 2.
- D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$.
- $P(k)$ is a permutation matrix as defined by $ipiv(k)$.
- $L(k)$ is a unit lower triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$$L(k) = \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & v & I \end{pmatrix}_{\begin{array}{c} k-1 \\ s \\ n-k-s+1 \end{array}}$$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(k+1:n,k)$.

If $s = 2$, the lower triangle of $D(k)$ overwrites $A(k,k)$, $A(k+1,k)$, and $A(k+1,k+1)$, and v overwrites $A(k+2:n,k:k+1)$.

See Also

[Matrix Storage Schemes](#)

?hetrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax**FORTRAN 77:**

```
call chetrf( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetrf( uplo, n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call hetrf( a [, uplo] [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>hetrf( int matrix_layout, char uplo, lapack_int n, <datatype>* a,
lapack_int lda, lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method:

```
if uplo='U', A = P*U*D*UH*PT
if uplo='L', A = P*L*D*LH*PT,
```

where A is the input matrix, P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE

This routine supports the Progress Routine feature. See [Progress Routine](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo

CHARACTER*1. Must be '`U`' or '`L`'.

Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If $uplo = 'U'$, the array a stores the upper triangular part of the matrix A , and A is factored as $P*U*D*U^H*P^T$.

If $uplo = 'L'$, the array a stores the lower triangular part of the matrix A , and A is factored as $P*L*D*L^H*P^T$.

n

INTEGER. The order of matrix A ; $n \geq 0$.

<i>a, work</i>	COMPLEX for chetrf DOUBLE COMPLEX for zhetrf. Arrays, DIMENSION $a(1:\text{lda}, :)$, $\text{work}(1:\text{lwork})$. The array <i>a</i> contains the upper or the lower triangular part of the matrix <i>A</i> (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array of dimension at least $\max(1, \text{lwork})$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array ($\text{lwork} \geq n$). If $\text{lwork} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See Application Notes for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	The upper or lower triangular part of <i>a</i> is overwritten by details of the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i> . If <i>ipiv</i> (<i>i</i>) = <i>k</i> > 0, then d_{ii} is a 1-by-1 block, and the <i>i</i> -th row and column of <i>A</i> was interchanged with the <i>k</i> -th row and column. If <i>uplo</i> = 'U' and <i>ipiv</i> (<i>i</i>) = <i>ipiv</i> (<i>i</i> -1) = <i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> -1, and the (<i>i</i> -1)-th row and column of <i>A</i> was interchanged with the <i>m</i> -th row and column. If <i>uplo</i> = 'L' and <i>ipiv</i> (<i>i</i>) = <i>ipiv</i> (<i>i</i> +1) = <i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and the (<i>i</i> +1)-th row and column of <i>A</i> was interchanged with the <i>m</i> -th row and column.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , d_{ii} is 0. The factorization has been completed, but <i>D</i> is exactly singular. Division by 0 will occur if you use <i>D</i> for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetrf` interface are as follows:

<code>a</code>	holds the matrix A of size (n, n)
<code>ipiv</code>	holds the vector of length n
<code>uplo</code>	must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

This routine is suitable for Hermitian matrices that are not known to be positive-definite. If A is in fact positive-definite, the routine does not perform interchanges, and no 2-by-2 diagonal blocks occur in D .

For better performance, try using $\text{lwork} = n * \text{blocksize}$, where blocksize is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of lwork for the first run or set $\text{lwork} = -1$.

If you choose the first option and set any of admissible lwork sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set $\text{lwork} = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set lwork to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the corresponding columns of the array `a`, but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $\text{ipiv}(i) = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array `a`.

If $\text{uplo} = \text{'U'}$, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$\|E\| \leq c(n)\varepsilon \|P\| \|U\| \|D\| \|U^T\| \|P^T\|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for the computed L and D when $\text{uplo} = \text{'L'}$.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

<code>?hetrs</code>	to solve $A^T X = B$
<code>?hecon</code>	to estimate the condition number of A
<code>?hetri</code>	to compute the inverse of A .

See Also

`mkl_progress`

`?hetrf_rook`

Computes the bounded Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax

FORTRAN 77:

```
call chetrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetrf_rook( uplo, n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call hetrf_rook( a [, uplo] [,ipiv] [,info] )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the factorization of a complex Hermitian matrix A using the bounded Bunch-Kaufman diagonal pivoting method:

```
if uplo='U', A = U*D*UH
if uplo='L', A = L*D*LH,
```

where A is the input matrix, U (or L) is a product of permutation and unit upper (or lower) triangular matrices, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored:
	If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A .
	If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A .
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a</i> , <i>work</i>	COMPLEX for chetrf_rook COMPLEX*16 for zhetrf_rook. Arrays, size (<i>lda</i> ,*), <i>work</i> (*).
	The array <i>a</i> contains the upper or the lower triangular part of the matrix A (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$. If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of <i>a</i> contains the upper triangular part of the matrix A , and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of <i>a</i> contains the lower triangular part of the matrix A , and the strictly upper triangular part of <i>a</i> is not referenced. <i>work</i> (*) is a workspace array of dimension at least $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array (<i>lwork</i> $\geq n$).

The length of `work`. $lwork \geq 1$. For best performance $lwork \geq n*nb$, where nb is the block size returned by `ilaenv`.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`.

Output Parameters

`a`

The block diagonal matrix D and the multipliers used to obtain the factor U or L (see Application Notes for further details).

`work(1)`

If $info = 0$, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`ipiv`

INTEGER.

Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D .

- If $uplo = 'U'$:

If $ipiv(k) > 0$, then rows and columns k and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, then rows and columns k and $-ipiv(k)$ were interchanged and rows and columns $k - 1$ and $-ipiv(k - 1)$ were interchanged, $D_{k-1:k,k-1:k}$ is a 2-by-2 diagonal block.

- If $uplo = 'L'$:

If $ipiv(k) > 0$, then rows and columns k and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns k and $-ipiv(k)$ were interchanged and rows and columns $k + 1$ and $-ipiv(k + 1)$ were interchanged, $D_{k:k+1,k:k+1}$ is a 2-by-2 diagonal block.

`info`

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, D_{ii} is exactly 0. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by 0 will occur if you use D for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetrf_rook` interface are as follows:

`a`

holds the matrix A of size (n, n)

<i>ipiv</i>	holds the vector of length <i>n</i>
<i>uplo</i>	must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *uplo* = 'U', then $A = U^*D*U^H$, where

$$U = P(n)*U(n)* \dots *P(k)U(k)* \dots,$$

i.e., U is a product of terms $P(k)*U(k)$, where k decreases from n to 1 in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$. $P(k)$ is a permutation matrix as defined by *ipiv*(k), and $U(k)$ is a unit upper triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$U(k) =$	$\begin{pmatrix} I & v & 0 & \dots \\ 0 & I & 0 & \dots \\ 0 & 0 & I & \dots \\ k-s & s & n-k & \dots \end{pmatrix}$	$k-s$
		s
		$n-k$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(1:k-1,k)$.

If $s = 2$, the upper triangle of $D(k)$ overwrites $A(k-1,k-1)$, $A(k-1,k)$, and $A(k,k)$, and v overwrites $A(1:k-2,k-1:k)$.

If *uplo* = 'L', then $A = L^*D*L^H$, where

$$L = P(1)*L(1)* \dots *P(k)*L(k)* \dots,$$

i.e., L is a product of terms $P(k)*L(k)$, where k increases from 1 to n in steps of 1 or 2, and D is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $D(k)$. $P(k)$ is a permutation matrix as defined by *ipiv*(k), and $L(k)$ is a unit lower triangular matrix, such that if the diagonal block $D(k)$ is of order s ($s = 1$ or 2), then

$L(k) =$	$\begin{pmatrix} I & 0 & 0 & \dots \\ 0 & I & 0 & \dots \\ 0 & v & I & \dots \\ k-1 & s & n-k-s+1 & \dots \end{pmatrix}$	$k-1$
		s
		$n-k-s+1$

If $s = 1$, $D(k)$ overwrites $A(k,k)$, and v overwrites $A(k+1:n,k)$.

If $s = 2$, the lower triangle of $D(k)$ overwrites $A(k,k)$, $A(k+1,k)$, and $A(k+1,k+1)$, and v overwrites $A(k+2:n,k:k+1)$.

See Also

[mkl_progress](#)

?sptrf

Computes the Bunch-Kaufman factorization of a symmetric matrix using packed storage.

Syntax

FORTRAN 77:

```
call ssptrf( uplo, n, ap, ipiv, info )
call dsptrf( uplo, n, ap, ipiv, info )
call csptrf( uplo, n, ap, ipiv, info )
call zsptrf( uplo, n, ap, ipiv, info )
```

Fortran 95:

```
call sptrf( ap [,uplo] [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>sptrf( int matrix_layout, char uplo, lapack_int n, <datatype>* ap, lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the factorization of a real/complex symmetric matrix A stored in the packed format using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

```
if uplo='U',  $A = P \cdot U \cdot D \cdot U^T \cdot P^T$   
if uplo='L',  $A = P \cdot L \cdot D \cdot L^T \cdot P^T$ ,
```

where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is packed in the array <i>ap</i> and how A is factored: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the matrix A , and A is factored as $P \cdot U \cdot D \cdot U^T \cdot P^T$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the matrix A , and A is factored as $P \cdot L \cdot D \cdot L^T \cdot P^T$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>ap</i>	REAL for ssptrf DOUBLE PRECISION for dsptrf COMPLEX for csptrf DOUBLE COMPLEX for zsptrf. Array, DIMENSION at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in <i>packed storage</i> (see Matrix Storage Schemes).

Output Parameters

<i>ap</i>	The upper or lower triangle of A (as specified by <i>uplo</i>) is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of D . If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the i -th row and column of A was interchanged with the k -th row and column. If <i>uplo</i> = 'U' and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and the $(i-1)$ -th row and column of A was interchanged with the m -th row and column. If <i>uplo</i> = 'L' and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and the $(i+1)$ -th row and column of A was interchanged with the m -th row and column.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the i -th parameter had an illegal value. If <i>info</i> = i , d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sptrf` interface are as follows:

<i>ap</i>	Holds the array A of size $(n^*(n+1)/2)$.
<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L overwrite elements of the corresponding columns of the matrix A , but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv(i) = i$ for all $i = 1 \dots n$, then all off-diagonal elements of U (L) are stored explicitly in packed form.

If *uplo* = 'U', the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$\|E\| \leq c(n)\varepsilon P\|U\|\|D\|\|U^T\|P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. A similar estimate holds for the computed L and D when *uplo* = 'L'.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

?sptrs	to solve $A^*X = B$
?spcon	to estimate the condition number of A
?sptri	to compute the inverse of A .

See Also

[mkl_progress](#)

?hpstrf

Computes the Bunch-Kaufman factorization of a complex Hermitian matrix using packed storage.

Syntax

FORTRAN 77:

```
call chptrf( uplo, n, ap, ipiv, info )
call zhptrf( uplo, n, ap, ipiv, info )
```

Fortran 95:

```
call hpstrf( ap [,uplo] [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>hpstrf( int matrix_layout, char uplo, lapack_int n, <datatype>* ap, lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the factorization of a complex Hermitian packed matrix A using the Bunch-Kaufman diagonal pivoting method:

```
if uplo='U', A = P*U*D*UH*PT
if uplo='L', A = P*L*D*LH*PT,
```

where A is the input matrix, P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo	CHARACTER*1. Must be 'U' or 'L'.
------	----------------------------------

Indicates whether the upper or lower triangular part of A is packed and how A is factored:

If $uplo = 'U'$, the array ap stores the upper triangular part of the matrix A , and A is factored as $P^*U*D*U^H*P^T$.

If $uplo = 'L'$, the array ap stores the lower triangular part of the matrix A , and A is factored as $P*L*D*L^H*P^T$.

n

INTEGER. The order of matrix A ; $n \geq 0$.

ap

COMPLEX for chptrf

DOUBLE COMPLEX for zhpstrf.

Array, DIMENSION at least $\max(1, n(n+1)/2)$. The array ap contains the upper or the lower triangular part of the matrix A (as specified by $uplo$) in *packed storage* (see [Matrix Storage Schemes](#)).

Output Parameters

ap

The upper or lower triangle of A (as specified by $uplo$) is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

$ipiv$

INTEGER.

Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of D . If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the i -th row and column of A was interchanged with the k -th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and the $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and the $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

$info$

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hptrf` interface are as follows:

ap

Holds the array A of size $(n*(n+1)/2)$.

$ipiv$

Holds the vector of length n .

`uplo` Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the corresponding columns of the array a , but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).

If $ipiv(i) = i$ for all $i = 1 \dots n$, then all off-diagonal elements of $U(L)$ are stored explicitly in the corresponding elements of the array a .

If $\text{uplo} = \text{'U'}$, the computed factors U and D are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n) \varepsilon P |U| |D| |U^T| P^T$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

A similar estimate holds for the computed L and D when `uplo = 'L'`.

The total number of floating-point operations is approximately $(4/3)n^3$.

After calling this routine, you can call the following routines:

?hpcon to estimate the cost.

?hptri to compute the inverse of A .

See Also

mkl_progress

Routines for Solving Systems of Linear Equations

This section describes the LAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see [Routines for Matrix Factorization](#) in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

sgetrs dgetrs cgetrs zgetrs Reference

Solves a system of linear equations with an LU-factored square matrix, with multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call dgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call cgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
call zgetrs( trans, n, nrhs, a, lda, ipiv, b, ldb, info )
```

Fortran 95:

```
call getrs( a, ipiv, b [, trans] [, info] )
```

C:

```
lapack_int LAPACKE_<?>getrs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const <datatype>* a, lapack_int lda, const lapack_int* ipiv, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the following systems of linear equations:

$$\begin{aligned} A^*X &= B && \text{if } trans = 'N', \\ A^T*X &= B && \text{if } trans = 'T', \\ A^H*X &= B && \text{if } trans = 'C' \text{ (for complex matrices only).} \end{aligned}$$

Before calling this routine, you must call [?getrf](#) to compute the LU factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'.
	Indicates the form of the equations:
	If <i>trans</i> = 'N', then $A^*X = B$ is solved for X .
	If <i>trans</i> = 'T', then $A^T*X = B$ is solved for X .
	If <i>trans</i> = 'C', then $A^H*X = B$ is solved for X .
<i>n</i>	INTEGER. The order of A ; the number of rows in B ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a</i> , <i>b</i>	REAL for sgetrs DOUBLE PRECISION for dgetrs COMPLEX for cgetrs DOUBLE COMPLEX for zgetrs. Arrays: $a(1:da, :)$, $b(1:db, :)$. The array <i>a</i> contains LU factorization of matrix A resulting from the call of ?getrf . The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of <i>a</i> must be at least $\max(1, n)$, the second dimension of <i>b</i> at least $\max(1, nrhs)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ipiv</i>	INTEGER.

Array, DIMENSION at least $\max(1, n)$. The *ipiv* array, as returned by [?getrf](#).

Output Parameters

b Overwritten by the solution matrix X .
info INTEGER. If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `getrs` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size (n, nrhs) .
<i>ipiv</i>	Holds the vector of length n .
<i>trans</i>	Must be ' <code>N</code> ', ' <code>C</code> ', or ' <code>T</code> '. The default value is ' <code>N</code> '.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n) \varepsilon P |L| |U|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

.....

where $\text{cond}(A, x) = \|A^{-1}\| |A| \|x\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?gecon](#).

To refine the solution and estimate the error, call `?gerfs`.

?gbtrs

Solves a system of linear equations with an LU-factored band matrix, with multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call dgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call cgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call zgbtrs( trans, n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
```

Fortran 95:

```
call gbtrs( ab, b, ipiv, [, kl] [, trans] [, info] )
```

C:

```
lapack_int LAPACKE_<?>gbtrs( int matrix_layout, char trans, lapack_int n, lapack_int
kl, lapack_int ku, lapack_int nrhs, const <datatype>* ab, lapack_int ldab,
const lapack_int* ipiv, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the following systems of linear equations:

$$\begin{aligned} A^*X = B && \text{if } trans='N', \\ A^T*X = B && \text{if } trans='T', \\ A^H*X = B && \text{if } trans='C' \text{ (for complex matrices only).} \end{aligned}$$

Here A is an LU -factored general band matrix of order n with kl non-zero subdiagonals and ku nonzero superdiagonals. Before calling this routine, call [?gbtrf](#) to compute the LU factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'.
<i>n</i>	INTEGER. The order of A ; the number of rows in B ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab</i> , <i>b</i>	REAL for sgbtrs DOUBLE PRECISION for dgbtrs COMPLEX for cgbtrs

DOUBLE COMPLEX for zgbtrs.

Arrays: $ab(1dab, *)$, $b(1db, *)$.

The array ab contains the matrix A in *band storage* (see [Matrix Storage Schemes](#)).

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

The second dimension of ab must be at least $\max(1, n)$, and the second dimension of b at least $\max(1, nrhs)$.

$1dab$

INTEGER. The leading dimension of the array ab ; $1dab \geq 2*kl + ku + 1$.

$1db$

INTEGER. The leading dimension of b ; $1db \geq \max(1, n)$.

$ipiv$

INTEGER. Array, DIMENSION at least $\max(1, n)$. The $ipiv$ array, as returned by [?gbtrf](#).

Output Parameters

b

Overwritten by the solution matrix X .

$info$

INTEGER. If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbtrs` interface are as follows:

ab

Holds the array A of size $(2*kl+ku+1, n)$.

b

Holds the matrix B of size $(n, nrhs)$.

$ipiv$

Holds the vector of length $\min(m, n)$.

kl

If omitted, assumed $kl = ku$.

ku

Restored as $1da-2*kl-1$.

$trans$

Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(kl + ku + 1)\varepsilon P|L||U|$$

$c(k)$ is a modest linear function of k , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\begin{array}{ccccccccc} & | & \dots & : & : & : & : & : & : \\ \dots & | & \dots \\ & | & \dots & : & : & : & : & : & : \end{array}$$

where $\text{cond}(A, x) = |||A^{-1}||A||x||_{\infty} / ||x||_{\infty} \leq ||A^{-1}||_{\infty} ||A||_{\infty} = \kappa_{\infty}(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_{\infty}(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_{\infty}(A)$.

The approximate number of floating-point operations for one right-hand side vector is $2n(ku + 2kl)$ for real flavors. The number of operations for complex flavors is 4 times greater. All these estimates assume that kl and ku are much less than $\min(m,n)$.

To estimate the condition number $\kappa_{\infty}(A)$, call [?gbcon](#).

To refine the solution and estimate the error, call [?gbrfs](#).

[?gttrs](#)

Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by [?gttrf](#).

Syntax

FORTRAN 77:

```
call sgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call dgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call cgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
call zgttrs( trans, n, nrhs, dl, d, du, du2, ipiv, b, ldb, info )
```

Fortran 95:

```
call gttrs( dl, d, du, du2, b, ipiv [, trans] [,info] )
```

C:

```
lapack_int LAPACKE_<?>gttrs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs,
const <datatype>* dl, const <datatype>* d, const <datatype>* du,
const <datatype>* du2, const lapack_int* ipiv, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the following systems of linear equations with multiple right hand sides:

$$\begin{aligned} A^*X = B && \text{if } trans='N', \\ A^T*X = B && \text{if } trans='T', \\ A^H*X = B && \text{if } trans='C' \text{ (for complex matrices only).} \end{aligned}$$

Before calling this routine, you must call [?gttrf](#) to compute the LU factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $A^X = B$ is solved for X . If <i>trans</i> = 'T', then $A^T X = B$ is solved for X . If <i>trans</i> = 'C', then $A^H X = B$ is solved for X .
<i>n</i>	INTEGER. The order of A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, that is, the number of columns in B ; $nrhs \geq 0$.
<i>dl, d, du, du2, b</i>	REAL for <code>sgttrs</code> DOUBLE PRECISION for <code>dgttrs</code> COMPLEX for <code>cgttrs</code> DOUBLE COMPLEX for <code>zgttrs</code> . Arrays: $dl(n - 1), d(n), du(n - 1), du2(n - 2), b(ldb, nrhs)$. The array <i>dl</i> contains the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A . The array <i>d</i> contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A . The array <i>du</i> contains the $(n - 1)$ elements of the first superdiagonal of U . The array <i>du2</i> contains the $(n - 2)$ elements of the second superdiagonal of U . The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION (<i>n</i>). The <i>ipiv</i> array, as returned by <code>?gttrf</code> .

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gttrs` interface are as follows:

<code>dl</code>	Holds the vector of length $(n-1)$.
<code>d</code>	Holds the vector of length n .
<code>du</code>	Holds the vector of length $(n-1)$.
<code>du2</code>	Holds the vector of length $(n-2)$.
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>ipiv</code>	Holds the vector of length n .
<code>trans</code>	Must be ' <code>N</code> ', ' <code>C</code> ', or ' <code>T</code> '. The default value is ' <code>N</code> '.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\epsilon P|L||U|$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\begin{array}{ccccccccc} & & | & . & : & : & : & : & \\ & & | & . & : & : & : & : & \\ \dots & \dots & | & . & : & : & : & : & \dots \\ & & | & . & : & : & : & : & \end{array}$$

where $\text{cond}(A, x) = ||A^{-1}||A||x||_{\infty}/||x||_{\infty} \leq ||A^{-1}||_{\infty} ||A||_{\infty} = \kappa_{\infty}(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_{\infty}(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_{\infty}(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $7n$ (including n divisions) for real flavors and $34n$ (including $2n$ divisions) for complex flavors.

To estimate the condition number $\kappa_{\infty}(A)$, call `?gtcon`.

To refine the solution and estimate the error, call `?gtrfs`.

?dttrsbs

Solves a system of linear equations with a diagonally dominant tridiagonal matrix using the LU factorization computed by `?dttrfb`.

Syntax

FORTRAN 77:

```
call sdtrrsb( trans, n, nrhs, dl, d, du, b, ldb, info )
call ddtrrsb( trans, n, nrhs, dl, d, du, b, ldb, info )
```

```
call cdttrsbs( trans, n, nrhs, dl, d, du, b, ldb, info )
call zdttrsbs( trans, n, nrhs, dl, d, du, b, ldb, info )
```

Fortran 95:

```
call dttrsbs( dl, d, du, b [, trans] [, info] )
```

C:

```
void sdttrsbs (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const float
* dl, const float * d, const float * du, float * b, const MKL_INT * ldb, MKL_INT *
info );
void ddttrsbs (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const double
* dl, const double * d, const double * du, double * b, const MKL_INT * ldb, MKL_INT *
info );
void cdttrsbs (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const
MKL_Complex8 * dl, const MKL_Complex8 * d, const MKL_Complex8 * du, MKL_Complex8 * b,
const MKL_INT * ldb, MKL_INT * info );
void zdttrsbs (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const
MKL_Complex16 * dl, const MKL_Complex16 * d, const MKL_Complex16 * du, MKL_Complex16 *
b, const MKL_INT * ldb, MKL_INT * info );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The ?dttrsbs routine solves the following systems of linear equations with multiple right hand sides for X :

$$\begin{aligned} A^*X = B & \quad \text{if } \text{trans} = 'N', \\ A^T*X = B & \quad \text{if } \text{trans} = 'T', \\ A^H*X = B & \quad \text{if } \text{trans} = 'C' \text{ (for complex matrices only).} \end{aligned}$$

Before calling this routine, call [?dttrfb](#) to compute the factorization of A .

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'.
Indicates the form of the equations solved for X :	
If <i>trans</i> = 'N', then $A^*X = B$.	
If <i>trans</i> = 'T', then $A^T*X = B$.	
If <i>trans</i> = 'C', then $A^H*X = B$.	
<i>n</i>	INTEGER. The order of A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, that is, the number of columns in B ; $nrhs \geq 0$.
<i>dl, d, du, b</i>	REAL for sdttrsbs

DOUBLE PRECISION for ddttrsbs

COMPLEX for cdttrsbs

DOUBLE COMPLEX for zdttrsbs.

Arrays: $d_1(n - 1)$, $d(n)$, $du(n - 1)$, $b(ldb, nrhs)$.

The array d_1 contains the $(n - 1)$ multipliers that define the matrices L_1, L_2 from the factorization of A .

The array d contains the n diagonal elements of the upper triangular matrix U from the factorization of A .

The array du contains the $(n - 1)$ elements of the superdiagonal of U .

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

ldb

INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b

Overwritten by the solution matrix X .

$info$

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

?potrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```
call spotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call dpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call cpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
call zpotrs( uplo, n, nrhs, a, lda, b, ldb, info )
```

Fortran 95:

```
call potrs( a, b [,uplo] [, info] )
```

C:

```
lapack_int LAPACKE_<?>potrs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const <datatype>* a, lapack_int lda, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the system of linear equations $A^*X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A , given the Cholesky factorization of A :

$$\begin{array}{ll} A = U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} & \text{if } uplo = 'U' \\ A = L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} & \text{if } uplo = 'L' \end{array}$$

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

Before calling this routine, you must call [potrf](#) to compute the Cholesky factorization of A .

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '.
	Indicates how the input matrix A has been factored:
	If <code>uplo</code> = ' <code>U</code> ', the upper triangle of A is stored.
	If <code>uplo</code> = ' <code>L</code> ', the lower triangle of A is stored.
<code>n</code>	INTEGER. The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
<code>a, b</code>	REAL for <code>sputrs</code> DOUBLE PRECISION for <code>dputrs</code> COMPLEX for <code>cputrs</code> DOUBLE COMPLEX for <code>zputrs</code> . Arrays: $a(1da, *)$, $b(1db, *)$. The array a contains the factor U or L (see <code>uplo</code>). The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of a must be at least $\max(1, n)$, the second dimension of b at least $\max(1, nrhs)$.
<code>lda</code>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<code>ldb</code>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

<code>b</code>	Overwritten by the solution matrix X .
<code>info</code>	INTEGER. If <code>info</code> = 0, the execution is successful. If <code>info</code> = $-i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `potrs` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

If $uplo = 'U'$, the computed solution for each right-hand side b is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |U^H| |U|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for $uplo = 'L'$. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\left| \begin{array}{c} | \\ | \\ \dots & \dots & \dots & \dots & \dots \\ | & | \end{array} \right|$$

where $\text{cond}(A, x) = |||A^{-1}|||A|||x|||_\infty / |||x|||_\infty \leq |||A^{-1}|||_\infty |||A|||_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$. The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?pocon](#).

To refine the solution and estimate the error, call [?porfs](#).

?pftrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite matrix using the Rectangular Full Packed (RFP) format.

Syntax

FORTRAN 77:

```
call spftrs( transr, uplo, n, nrhs, a, b, ldb, info )
call dpftrs( transr, uplo, n, nrhs, a, b, ldb, info )
call cpftrs( transr, uplo, n, nrhs, a, b, ldb, info )
call zpftrs( transr, uplo, n, nrhs, a, b, ldb, info )
```

C:

```
lapack_int LAPACKE_<?>pftrs( int matrix_layout, char transr, char uplo, lapack_int n,
lapack_int nrhs, const <datatype>* a, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`

- C: mkl.h

Description

The routine solves a system of linear equations $A \cdot X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A using the Cholesky factorization of A :

$$\begin{array}{ll} A = U^T \cdot U \text{ for real data, } A = U^H \cdot U \text{ for complex data} & \text{if } \text{uplo} = 'U' \\ A = L \cdot L^T \text{ for real data, } A = L \cdot L^H \text{ for complex data} & \text{if } \text{uplo} = 'L' \end{array}$$

computed by [?pftrf](#). L stands for a lower triangular matrix and U - for an upper triangular matrix.

The matrix A is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>transr</i>	CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data). If <i>transr</i> = 'N', the Normal <i>transr</i> of RFP A is stored. If <i>transr</i> = 'T', the Transpose <i>transr</i> of RFP A is stored. If <i>transr</i> = 'C', the Conjugate-Transpose <i>transr</i> of RFP A is stored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of the RFP matrix A is stored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A . If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix B ; $nrhs \geq 0$.
<i>a, b</i>	REAL for <i>sfptrs</i> DOUBLE PRECISION for <i>dpftrs</i> COMPLEX for <i>cpftrs</i> DOUBLE COMPLEX for <i>zpftrs</i> . Arrays: $a(n \cdot (n+1)/2)$, $b(ldb, nrhs)$. The array <i>a</i> contains the matrix A in the RFP format. The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	The solution matrix X .
<i>info</i>	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

?pptrs

Solves a system of linear equations with a packed Cholesky-factored symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```
call spptrs( uplo, n, nrhs, ap, b, ldb, info )
call dpptrs( uplo, n, nrhs, ap, b, ldb, info )
call cpptrs( uplo, n, nrhs, ap, b, ldb, info )
call zpptrs( uplo, n, nrhs, ap, b, ldb, info )
```

Fortran 95:

```
call pptrs( ap, b [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>pptrs( int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const <datatype>* ap, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the system of linear equations $A^*X = B$ with a packed symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A , given the Cholesky factorization of A :

$$\begin{array}{ll} A = U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} & \text{if } uplo='U' \\ A = L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} & \text{if } uplo='L' \end{array}$$

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

Before calling this routine, you must call `?pptrf` to compute the Cholesky factorization of A .

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored:
-------------	---

	If $uplo = 'U'$, the upper triangle of A is stored.
	If $uplo = 'L'$, the lower triangle of A is stored.
n	INTEGER. The order of matrix A ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides ($nrhs \geq 0$).
ap, b	REAL for spptrs DOUBLE PRECISION for dpptrs COMPLEX for cpptrs DOUBLE COMPLEX for zpptrs. Arrays: $ap(*)$, $b(ldb,*)$
	The dimension of ap must be at least $\max(1, n(n+1)/2)$.
	The array ap contains the factor U or L , as specified by $uplo$, in packed storage (see Matrix Storage Schemes).
	The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.
ldb	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b	Overwritten by the solution matrix X .
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pptrs` interface are as follows:

ap	Holds the array A of size $(n*(n+1)/2)$.
b	Holds the matrix B of size $(n, nrhs)$.
$uplo$	Must be ' U ' or ' L '. The default value is ' U '.

Application Notes

If $uplo = 'U'$, the computed solution for each right-hand side b is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |U^H| |U|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

A similar estimate holds for $uplo = 'L'$.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\begin{array}{ccccccccc} & & | & \dots & | & \dots & : & : & : \\ & & | & \dots & | & \dots & . & . & . \end{array}$$

where $\text{cond}(A, x) = |||A^{-1}|||A|||x|||_\infty / |||x|||_\infty \leq |||A^{-1}|||_\infty |||A|||_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?ppcon](#).

To refine the solution and estimate the error, call [?pprfs](#).

?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite band matrix.

Syntax

FORTRAN 77:

```
call spbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call dpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call cpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call zpbtrs( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
```

Fortran 95:

```
call pbtrs( ab, b [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>pbtrs( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
lapack_int nrhs, const <datatype>* ab, lapack_int ldab, <datatype>* b, lapack_int
ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for real data a system of linear equations $A \cdot X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite *band* matrix A , given the Cholesky factorization of A :

$$\begin{array}{ll} A = U^T \cdot U \text{ for real data, } A = U^H \cdot U \text{ for complex data} & \text{if } \text{uplo} = 'U' \\ A = L \cdot L^T \text{ for real data, } A = L \cdot L^H \text{ for complex data} & \text{if } \text{uplo} = 'L' \end{array}$$

where L is a lower triangular matrix and U is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

Before calling this routine, you must call [?pbtrf](#) to compute the Cholesky factorization of A in the band storage form.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix <i>A</i> has been factored:
	If <i>uplo</i> = 'U', the upper triangular factor is stored in <i>ab</i> .
	If <i>uplo</i> = 'L', the lower triangular factor is stored in <i>ab</i> .
<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix <i>A</i> ; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab, b</i>	REAL for spbtrs DOUBLE PRECISION for dpbtrs COMPLEX for cpbtrs DOUBLE COMPLEX for zpbtrs. Arrays: <i>ab</i> (<i>ldab</i> , *), <i>b</i> (<i>ldb</i> , *). The array <i>ab</i> contains the Cholesky factor, as returned by the factorization routine, in <i>band storage</i> form. The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>ab</i> must be at least $\max(1, n)$, and the second dimension of <i>b</i> at least $\max(1, nrhs)$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; $ldab \geq kd + 1$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbtrs` interface are as follows:

<i>ab</i>	Holds the array <i>A</i> of size (<i>kd</i> +1, <i>n</i>).
-----------	--

<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For each right-hand side *b*, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(kd + 1)\epsilon P|U^H||U| \text{ or } |E| \leq c(kd + 1)\epsilon P|L^H||L|$$

$c(k)$ is a modest linear function of k , and ϵ is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kd + 1) \operatorname{cond}(A, x)\epsilon$$

where $\operatorname{cond}(A, x) = |||A^{-1}||A||x||_\infty / ||x||_\infty \leq ||A^{-1}||_\infty ||A||_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector is $4n*kd$ for real flavors and $16n*kd$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?pbcon](#).

To refine the solution and estimate the error, call [?pbrfs](#).

?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal matrix using the factorization computed by [?pttrf](#).

Syntax

FORTRAN 77:

```
call spttrs( n, nrhs, d, e, b, ldb, info )
call dpttrs( n, nrhs, d, e, b, ldb, info )
call cpttrs( uplo, n, nrhs, d, e, b, ldb, info )
call zpttrs( uplo, n, nrhs, d, e, b, ldb, info )
```

Fortran 95:

```
call pttrs( d, e, b [,info] )
call pttrs( d, e, b [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_spttrs( int matrix_layout, lapack_int n, lapack_int nrhs,
                           const float* d, const float* e, float* b, lapack_int ldb );
lapack_int LAPACKE_dpttrs( int matrix_layout, lapack_int n, lapack_int nrhs,
                           const double* d, const double* e, double* b, lapack_int ldb );
```

```
lapack_int LAPACKE_cpttrs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* d, const lapack_complex_float* e, lapack_complex_float* b, lapack_int ldb );

lapack_int LAPACKE_zpttrs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* d, const lapack_complex_double* e, lapack_complex_double* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X a system of linear equations $A \cdot X = B$ with a symmetric (Hermitian) positive-definite tridiagonal matrix A . Before calling this routine, call [?pttrf](#) to compute the $L \cdot D \cdot L'$ for real data and the $L \cdot D \cdot L'$ or $U' \cdot D \cdot U$ factorization of A for complex data.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Used for cpttrs/zpttrs only. Must be 'U' or 'L'. Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and how A is factored: If <i>uplo</i> = 'U', the array <i>e</i> stores the superdiagonal of A , and A is factored as $U' \cdot D \cdot U$. If <i>uplo</i> = 'L', the array <i>e</i> stores the subdiagonal of A , and A is factored as $L \cdot D \cdot L'$.
<i>n</i>	INTEGER. The order of A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix B ; $nrhs \geq 0$.
<i>d</i>	REAL for spttrs, cpttrs DOUBLE PRECISION for dpttrs, zpttrs. Array, dimension (<i>n</i>). Contains the diagonal elements of the diagonal matrix D from the factorization computed by ?pttrf .
<i>e, b</i>	REAL for spttrs DOUBLE PRECISION for dpttrs COMPLEX for cpttrs DOUBLE COMPLEX for zpttrs. Arrays: $e(n - 1)$, $b(lsb, nrhs)$. The array <i>e</i> contains the (<i>n</i> - 1) off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by ?pttrf (see <i>uplo</i>).

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

ldb INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix X .

$info$ INTEGER. If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pttrs` interface are as follows:

d Holds the vector of length n .

e Holds the vector of length $(n-1)$.

b Holds the matrix B of size $(n, nrhs)$.

$uplo$ Used in complex flavors only. Must be '`U`' or '`L`'. The default value is '`U`'.

?sytrs

Solves a system of linear equations with a UDU- or LDL-factored symmetric matrix.

Syntax

FORTRAN 77:

```
call ssytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call dsytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call csytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zsytrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
```

Fortran 95:

```
call sytrs( a, b, ipiv [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>sytrs( int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const <datatype>* a, lapack_int lda, const lapack_int* ipiv, <datatype>* b,
lapack_int ldb );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine solves for X the system of linear equations $A^T X = B$ with a symmetric matrix A , given the Bunch-Kaufman factorization of A :

```
if uplo='U',           A = P*U*D*U^T*P^T
if uplo='L',           A = P*L*D*L^T*P^T,
```

where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply to this routine the factor U (or L) and the array $ipiv$ returned by the factorization routine [?sytrf](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular factor U of the factorization $A = P*U*D*U^T*P^T$.
	If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular factor L of the factorization $A = P*L*D*L^T*P^T$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sytrf .
<i>a, b</i>	REAL for ssytrs DOUBLE PRECISION for dsytrs COMPLEX for csytrs DOUBLE COMPLEX for zsytrs. Arrays: <i>a</i> (<i>lda</i> , *), <i>b</i> (<i>ldb</i> , *). The array <i>a</i> contains the factor U or L (see <i>uplo</i>). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of <i>a</i> must be at least $\max(1, n)$, and the second dimension of <i>b</i> at least $\max(1, nrhs)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytrs` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\epsilon P|U||D||U^T|P^T \text{ or } |E| \leq c(n)\epsilon P|L||D||U^T|P^T$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x)\epsilon$$

where $\operatorname{cond}(A, x) = |||A^{-1}|||A|||x|||_\infty / |||x|||_\infty \leq |||A^{-1}|||_\infty |||A|||_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?sycon](#).

To refine the solution and estimate the error, call [?syrfs](#).

?sytrs_rook

Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix.

Syntax

FORTRAN 77:

```
call ssytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
```

```
call dsytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call csytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zsytrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
```

Fortran 95:

```
call sytrs_rook( a, b, ipiv [,uplo] [,info] )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves a system of linear equations $A^T X = B$ with a symmetric matrix A , using the factorization $A = U^T D U$ or $A = L^T D L$ computed by [?sytrf_rook](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <i>uplo</i> = 'U', the factorization is of the form $A = U^T D U$.
	If <i>uplo</i> = 'L', the factorization is of the form $A = L^T D L$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sytrf_rook .
<i>a, b</i>	REAL for ssytrs_rook DOUBLE PRECISION for dsytrs_rook COMPLEX for csytrs_rook DOUBLE COMPLEX for zsytrs_rook. Arrays: <i>a</i> (<i>lda</i> , *), <i>b</i> (<i>ldb</i> , *). The array <i>a</i> contains the block diagonal matrix D and the multipliers used to obtain U or L as computed by ?sytrf_rook (see <i>uplo</i>). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of <i>a</i> must be at least $\max(1, n)$, and the second dimension of <i>b</i> at least $\max(1, nrhs)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytrs_rook` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?hetrs

Solves a system of linear equations with a UDU- or LDL-factored Hermitian matrix.

Syntax

FORTRAN 77:

```
call chetrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zhetrs( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
```

Fortran 95:

```
call hetrs( a, b, ipiv [, uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>hetrs( int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const <datatype>* a, lapack_int lda, const lapack_int* ipiv, <datatype>* b,
lapack_int ldb );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine solves for X the system of linear equations $A^*X = B$ with a Hermitian matrix A , given the Bunch-Kaufman factorization of A :

```
if uplo = 'U'           A = P*U*D*UH*PT
if uplo = 'L'           A = P*L*D*LH*PT,
```

where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply to this routine the factor U (or L) and the array *ipiv* returned by the factorization routine [?hetrf](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular factor U of the factorization $A = P*U*D*UH*PT$.
	If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular factor L of the factorization $A = P*L*D*LH*PT$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hetrf .
<i>a, b</i>	COMPLEX for chetrs DOUBLE COMPLEX for zhetrs. Arrays: <i>a</i> (<i>lda</i> , *), <i>b</i> (<i>ldb</i> , *). The array <i>a</i> contains the factor U or L (see <i>uplo</i>). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of <i>a</i> must be at least $\max(1, n)$, the second dimension of <i>b</i> at least $\max(1, nrhs)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetrs` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\epsilon P|U||D||U^H|P^T \text{ or } |E| \leq c(n)\epsilon P|L||D||L^H|P^T$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\begin{array}{ccccccccc} & & & & \dots & & & & \dots \\ & & & & \vdots & & & & \vdots \\ & & & & \dots & & & & \dots \\ & & & & \dots & & & & \dots \\ & & & & \vdots & & & & \vdots \\ & & & & \dots & & & & \dots \\ & & & & \vdots & & & & \vdots \end{array}$$

where $\text{cond}(A, x) = ||A^{-1}|||A|||x||_{\infty}/||x||_{\infty} \leq ||A^{-1}||_{\infty}||A||_{\infty} = \kappa_{\infty}(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_{\infty}(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$.

To estimate the condition number $\kappa_{\infty}(A)$, call [?hecon](#).

To refine the solution and estimate the error, call [?herfs](#).

?hetrs_rook

Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix.

Syntax

FORTRAN 77:

```
call chetrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
call zhetrs_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, info )
```

Fortran 95:

```
call hetrs_rook( a, b, ipiv [, uplo] [,info] )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for a system of linear equations $A^*X = B$ with a complex Hermitian matrix A using the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ computed by [?hetrf_rook](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <i>uplo</i> = 'U', the factorization is of the form $A = U^*D^*U^H$.
	If <i>uplo</i> = 'L', the factorization is of the form $A = L^*D^*L^H$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hetrf_rook .
<i>a, b</i>	COMPLEX for chetrs_rook DOUBLE COMPLEX for zhetrs_rook. Arrays: <i>a</i> (<i>lda</i> , *), <i>b</i> (<i>ldb</i> , *). The array <i>a</i> contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?hetrf_rook (see <i>uplo</i>). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of <i>a</i> must be at least $\max(1, n)$, the second dimension of <i>b</i> at least $\max(1, nrhs)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetrs_rook` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

?sytrs2

Solves a system of linear equations with a UDU - or LDL -factored symmetric matrix computed by `?sytrf` and converted by `?syconv`.

Syntax

FORTRAN 77:

```
call ssytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call dsytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call csytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call zsytrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
```

Fortran 95:

```
call sytrs2( a,b,ipiv[,uplo][,info] )
```

C:

```
lapack_int LAPACKE_<?>sytrs2( int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const <datatype>* a, lapack_int lda, const lapack_int* ipiv, <datatype>* b,
lapack_int ldb );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine solves a system of linear equations $A \cdot X = B$ with a symmetric matrix A using the factorization of A :

```
if uplo='U',           A = U*D*UT
if uplo='L',           A = L*D*LT
```

where

- U and L are upper and lower triangular matrices with unit diagonal
- D is a symmetric block-diagonal matrix.

The factorization is computed by `?sytrf` and converted by `?syconv`.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates how the input matrix <i>A</i> has been factored:
	If <i>uplo</i> = ' <i>U</i> ', the array <i>a</i> stores the upper triangular factor <i>U</i> of the factorization $A = U*D*U^T$.
	If <i>uplo</i> = ' <i>L</i> ', the array <i>a</i> stores the lower triangular factor <i>L</i> of the factorization $A = L*D*L^T$.
<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a, b</i>	REAL for <code>ssytrs2</code> DOUBLE PRECISION for <code>dsytrs2</code> COMPLEX for <code>csytrs2</code> DOUBLE COMPLEX for <code>zsytrs2</code> Arrays: <i>a</i> (<i>lda</i> ,*), <i>b</i> (<i>ldb</i> ,*).
	The array <i>a</i> contains the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> as computed by <code>?sytrf</code> .
	The array <i>b</i> contains the right-hand side matrix <i>B</i> .
	The second dimension of <i>a</i> must be at least <code>max(1, n)</code> , and the second dimension of <i>b</i> at least <code>max(1, nrhs)</code> .
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array of DIMENSION <i>n</i> . The <i>ipiv</i> array contains details of the interchanges and the block structure of <i>D</i> as determined by <code>?sytrf</code> .
<i>work</i>	REAL for <code>ssytrs2</code> DOUBLE PRECISION for <code>dsytrs2</code> COMPLEX for <code>csytrs2</code> DOUBLE COMPLEX for <code>zsytrs2</code> Workspace array, DIMENSION <i>n</i> .

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytrs2` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Indicates how the input matrix A has been factored. Must be ' <code>U</code> ' or ' <code>L</code> '.

See Also

[?sytrf](#)
[?syconv](#)

?hetrs2

Solves a system of linear equations with a UDU - or LDL -factored Hermitian matrix computed by `?hetrf` and converted by `?syconv`.

Syntax

FORTRAN 77:

```
call chetrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
call zhetrs2( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, info )
```

Fortran 95:

```
call hetrs2( a, b, ipiv [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>hetrs2( int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, const <datatype>* a, lapack_int lda, const lapack_int* ipiv, <datatype>* b,
lapack_int ldb );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine solves a system of linear equations $A^*X = B$ with a complex Hermitian matrix A using the factorization of A :

```
if uplo='U',           A = U*D*UH
if uplo='L',           A = L*D*LH
```

where

- U and L are upper and lower triangular matrices with unit diagonal
- D is a Hermitian block-diagonal matrix.

The factorization is computed by `?hetrf` and converted by `?syconv`.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '.
	Indicates how the input matrix A has been factored:
	If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = U \cdot D \cdot U^H$.
	If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = L \cdot D \cdot L^H$.
<code>n</code>	INTEGER. The order of matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<code>a, b</code>	COMPLEX for <code>chetrs2</code> DOUBLE COMPLEX for <code>zhetrs2</code> Arrays: $a(1:da, :)$, $b(1:db, :)$.
	The array a contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by <code>?hetrf</code> .
	The array b contains the right-hand side matrix B .
	The second dimension of a must be at least $\max(1, n)$, and the second dimension of b at least $\max(1, nrhs)$.
<code>lda</code>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<code>ldb</code>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.
<code>ipiv</code>	INTEGER. Array of DIMENSION n . The $ipiv$ array contains details of the interchanges and the block structure of D as determined by <code>?hetrf</code> .
<code>work</code>	COMPLEX for <code>chetrs2</code> DOUBLE COMPLEX for <code>zhetrs2</code> Workspace array, DIMENSION n .

Output Parameters

<code>b</code>	Overwritten by the solution matrix X .
<code>info</code>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetrs2` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

See Also

[?hetrf](#)

[?syconv](#)

[?sptrs](#)

Solves a system of linear equations with a UDU- or LDL-factored symmetric matrix using packed storage.

Syntax

FORTRAN 77:

```
call ssptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dsptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call csptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zsptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call sptrs( ap, b, ipiv [, uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>sptrs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const <datatype>* ap, const lapack_int* ipiv, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine solves for X the system of linear equations $A^T X = B$ with a symmetric matrix A , given the Bunch-Kaufman factorization of A :

`if uplo='U',` $A = PUDU^TP^T$

`if uplo='L',` $A = PLDL^TP^T,$

where P is a permutation matrix, U and L are upper and lower *packed* triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B . You must supply the factor U (or L) and the array $ipiv$ returned by the factorization routine [?sptrf](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <i>uplo</i> = 'U', the array <i>ap</i> stores the packed factor U of the factorization $A = P^*U^*D^*U^T*P^T$. If <i>uplo</i> = 'L', the array <i>ap</i> stores the packed factor L of the factorization $A = P^*L^*D^*L^T*P^T$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sptrf .
<i>ap, b</i>	REAL for ssptrs DOUBLE PRECISION for dsptrs COMPLEX for csptrs DOUBLE COMPLEX for zsptrs. Arrays: <i>ap</i> (*), <i>b</i> (<i>ldb</i> ,*).
	The dimension of <i>ap</i> must be at least $\max(1,n(n+1)/2)$. The array <i>ap</i> contains the factor U or L , as specified by <i>uplo</i> , in <i>packed storage</i> (see Matrix Storage Schemes). The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sptrs` interface are as follows:

<code>ap</code>	Holds the array A of size $(n^*(n+1)/2)$.
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^T|P^T \text{ or } |E| \leq c(n)\varepsilon P|L||D||L^T|P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x)\varepsilon$$

where $\operatorname{cond}(A, x) = |||A^{-1}||A||x||||_\infty / ||x||_\infty \leq ||A^{-1}||_\infty ||A||_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $2n^2$ for real flavors or $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?spcon](#).

To refine the solution and estimate the error, call [?sprfs](#).

?hptrs

Solves a system of linear equations with a UDU- or LDL-factored Hermitian matrix using packed storage.

Syntax

FORTRAN 77:

```
call chptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhptrs( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call hptrs( ap, b, ipiv [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>hptrs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const <datatype>* ap, const lapack_int* ipiv, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`

- C: mkl.h

Description

The routine solves for X the system of linear equations $A^*X = B$ with a Hermitian matrix A , given the Bunch-Kaufman factorization of A :

```
if uplo='U',           A = P*U*D*UH*PT
if uplo='L',           A = P*L*D*LH*PT,
```

where P is a permutation matrix, U and L are upper and lower *packed* triangular matrices with unit diagonal, and D is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix B .

You must supply to this routine the arrays ap (containing U or L) and $ipiv$ in the form returned by the factorization routine [?hptrf](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <i>uplo</i> = 'U', the array ap stores the packed factor U of the factorization $A = P*U*D*UH*PT$. If <i>uplo</i> = 'L', the array ap stores the packed factor L of the factorization $A = P*L*D*LH*PT$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hptrf .
<i>ap, b</i>	COMPLEX for chptrs DOUBLE COMPLEX for zhptrs. Arrays: $ap(*)$, $b(ldb, *)$.
	The dimension of ap must be at least $\max(1, n(n+1)/2)$. The array ap contains the factor U or L , as specified by <i>uplo</i> , in <i>packed storage</i> (see Matrix Storage Schemes). The array b contains the matrix B whose columns are the right-hand sides for the system of equations. The second dimension of b must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hptrs` interface are as follows:

<code>ap</code>	Holds the array A of size $(n^*(n+1)/2)$.
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon P|U||D||U^H|P^T \text{ or } |E| \leq c(n)\varepsilon P|L||D||L^H|P^T$$

$c(n)$ is a modest linear function of n , and ε is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x)\varepsilon$$

where $\operatorname{cond}(A, x) = |||A^{-1}|||A|||x|||_\infty / |||x|||_\infty \leq |||A^{-1}|||_\infty |||A|||_\infty = \kappa_\infty(A)$.

Note that $\operatorname{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?hpcon](#).

To refine the solution and estimate the error, call [?hprfs](#).

?trtrs

Solves a system of linear equations with a triangular matrix, with multiple right-hand sides.

Syntax

FORTRAN 77:

```
call strtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call dtrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call ctrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
call ztrtrs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, info )
```

Fortran 95:

```
call trtrs( a, b [,uplo] [, trans] [,diag] [,info] )
```

C:

```
lapack_int LAPACKE_<?>trtrs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const <datatype>* a, lapack_int lda, <datatype>* b,
lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the following systems of linear equations with a triangular matrix A , with multiple right-hand sides stored in B :

$$\begin{aligned} A^*X = B && \text{if } \text{trans} = 'N', \\ A^T X = B && \text{if } \text{trans} = 'T', \\ A^H X = B && \text{if } \text{trans} = 'C' \text{ (for complex matrices only).} \end{aligned}$$

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <code>uplo</code> = 'U', then A is upper triangular. If <code>uplo</code> = 'L', then A is lower triangular.
<code>trans</code>	CHARACTER*1. Must be 'N' or 'T' or 'C'. If <code>trans</code> = 'N', then $A^*X = B$ is solved for X . If <code>trans</code> = 'T', then $A^T X = B$ is solved for X . If <code>trans</code> = 'C', then $A^H X = B$ is solved for X .
<code>diag</code>	CHARACTER*1. Must be 'N' or 'U'. If <code>diag</code> = 'N', then A is not a unit triangular matrix. If <code>diag</code> = 'U', then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array <code>a</code> .
<code>n</code>	INTEGER. The order of A ; the number of rows in B ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<code>a, b</code>	REAL for strtrs DOUBLE PRECISION for dtrtrs COMPLEX for ctrtrs

DOUBLE COMPLEX for ztrtrs.

Arrays: $a(1da, *)$, $b(1db, *)$.

The array a contains the matrix A .

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

The second dimension of a must be at least $\max(1, n)$, the second dimension of b at least $\max(1, nrhs)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldb INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the solution matrix X .

$info$ INTEGER. If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trtrs` interface are as follows:

a Stands for argument `ap` in FORTRAN 77 interface. Holds the matrix A of size $(n * (n+1) / 2)$.

b Holds the matrix B of size $(n, nrhs)$.

$uplo$ Must be '`U`' or '`L`'. The default value is '`U`'.

$trans$ Must be '`N`', '`C`', or '`T`'. The default value is '`N`'.

$diag$ Must be '`N`' or '`U`'. The default value is '`N`'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |A|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \operatorname{cond}(A, x)\varepsilon \text{ provided } c(n) \operatorname{cond}(A, x)\varepsilon < 1$$

where $\text{cond}(A, x) = |||A^{-1}|||A|||x|||_\infty / |||x|||_\infty \leq |||A^{-1}|||_\infty |||A|||_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?trcon](#).

To estimate the error in the solution, call [?trrfs](#).

[?tptrs](#)

Solves a system of linear equations with a packed triangular matrix, with multiple right-hand sides.

Syntax

FORTRAN 77:

```
call stptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call dptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call cptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
call zptrs( uplo, trans, diag, n, nrhs, ap, b, ldb, info )
```

Fortran 95:

```
call tptrs( ap, b [,uplo] [, trans] [,diag] [,info] )
```

C:

```
lapack_int LAPACKE_<?>tptrs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const <datatype>* ap, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the following systems of linear equations with a packed triangular matrix A , with multiple right-hand sides stored in B :

$$\begin{aligned} A^*X &= B && \text{if } \text{trans} = 'N', \\ A^T X &= B && \text{if } \text{trans} = 'T', \\ A^H X &= B && \text{if } \text{trans} = 'C' \text{ (for complex matrices only).} \end{aligned}$$

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

`uplo` CHARACTER*1. Must be 'U' or 'L'.

Indicates whether A is upper or lower triangular:

	If <i>uplo</i> = 'U', then <i>A</i> is upper triangular.
	If <i>uplo</i> = 'L', then <i>A</i> is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'.
	If <i>trans</i> = 'N', then <i>A</i> * <i>X</i> = <i>B</i> is solved for <i>X</i> .
	If <i>trans</i> = 'T', then <i>A</i> ^T * <i>X</i> = <i>B</i> is solved for <i>X</i> .
	If <i>trans</i> = 'C', then <i>A</i> ^H * <i>X</i> = <i>B</i> is solved for <i>X</i> .
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'.
	If <i>diag</i> = 'N', then <i>A</i> is not a unit triangular matrix.
	If <i>diag</i> = 'U', then <i>A</i> is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	INTEGER. The order of <i>A</i> ; the number of rows in <i>B</i> ; <i>n</i> ≥ 0.
<i>nrhs</i>	INTEGER. The number of right-hand sides; <i>nrhs</i> ≥ 0.
<i>ap, b</i>	REAL for stptrs DOUBLE PRECISION for dtptrs COMPLEX for ctptrs DOUBLE COMPLEX for ztptrs. Arrays: <i>ap</i> (*), <i>b</i> (<i>ldb</i> ,*).
	The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the matrix <i>A</i> in <i>packed storage</i> (see Matrix Storage Schemes).
	The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the system of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; <i>ldb</i> ≥ $\max(1, n)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tptrs` interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.

<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\epsilon |A|$$

$c(n)$ is a modest linear function of n , and ϵ is the machine precision.

If x_0 is the true solution, the computed solution x satisfies this error bound:

$$\begin{array}{ccccccccc} & | & \dots & : & : & : & : & : & : \\ & | & \dots & : & : & : & : & : & : \\ & | & \dots & : & : & : & : & : & : \\ & | & \dots & : & : & : & : & : & : \\ & | & \dots & : & : & : & : & : & : \\ & | & \dots & : & : & : & : & : & : \\ & | & \dots & : & : & : & : & : & : \\ & | & \dots & : & : & : & : & : & : \end{array}$$

where $\text{cond}(A, x) = ||A^{-1}||\|A\| \|x\|_\infty / \|x\|_\infty \leq ||A^{-1}||_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is n^2 for real flavors and $4n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?tpcon](#).

To estimate the error in the solution, call [?tprrfs](#).

?tbtrs

Solves a system of linear equations with a band triangular matrix, with multiple right-hand sides.

Syntax

FORTRAN 77:

```
call stbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call dtbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call ctbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
call ztbtrs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, info )
```

Fortran 95:

```
call tbtrs( ab, b [, uplo] [, trans] [, diag] [, info] )
```

C:

```
lapack_int LAPACKE_<?>tbtrs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int kd, lapack_int nrhs, const <datatype>* ab, lapack_int ldab,
<datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the following systems of linear equations with a band triangular matrix A , with multiple right-hand sides stored in B :

$$\begin{aligned} A^*X = B & \quad \text{if } trans='N', \\ A^T*X = B & \quad \text{if } trans='T', \\ A^H*X = B & \quad \text{if } trans='C' \text{ (for complex matrices only).} \end{aligned}$$

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. If <i>trans</i> = 'N', then $A^*X = B$ is solved for X . If <i>trans</i> = 'T', then $A^T*X = B$ is solved for X . If <i>trans</i> = 'C', then $A^H*X = B$ is solved for X .
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ab</i> .
<i>n</i>	INTEGER. The order of A ; the number of rows in B ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab, b</i>	REAL for stbtrs DOUBLE PRECISION for dtbtrs COMPLEX for ctbtrs DOUBLE COMPLEX for ztbtrs. Arrays: <i>ab</i> (<i>ldab</i> , *), <i>b</i> (<i>ldb</i> , *).

The array ab contains the matrix A in *band storage form*.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

The second dimension of ab must be at least $\max(1, n)$, the second dimension of b at least $\max(1, \text{nrhs})$.

1dab

INTEGER. The leading dimension of ab ; $ldab \geq kd + 1$.

1db

INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

b

Overwritten by the solution matrix X .

info

INTEGER. If *info*=0, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tbtrs` interface are as follows:

ab

Holds the array A of size $(kd+1, n)$

b

Holds the matrix B of size (n, nrhs) .

uplo

Must be 'U' or 'L'. The default value is 'U'.

trans

Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

For each right-hand side b , the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n) s |\Lambda|$$

$c(n)$ is a modest linear function of n , and ε is the machine precision. If x_0 is the true solution, the computed solution x satisfies this error bound:

.....

where $\text{cond}(A, x) = \|\|A^{-1}\|A\| |x|\|_\infty / \|x\|_\infty \leq \|A^{-1}\|_\infty \|A\|_\infty = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$; the condition number of A^T and A^H might or might not be equal to $\kappa_\infty(A)$.

The approximate number of floating-point operations for one right-hand side vector b is $2n*kd$ for real flavors and $8n*kd$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call [?tbcon](#).

To estimate the error in the solution, call [?tbrfs](#).

Routines for Estimating the Condition Number

This section describes the LAPACK routines for estimating the *condition number* of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations (see [Error Analysis](#)). Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

[?gecon](#)

Estimates the reciprocal of the condition number of a general matrix in the 1-norm or the infinity-norm.

Syntax

FORTRAN 77:

```
call sgecon( norm, n, a, lda, anorm, rcond, work, iwork, info )
call dgecon( norm, n, a, lda, anorm, rcond, work, iwork, info )
call cgecon( norm, n, a, lda, anorm, rcond, work, rwork, info )
call zgecon( norm, n, a, lda, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call gecon( a, anorm, rcond [,norm] [,info] )
```

C:

```
lapack_int LAPACKE_sgecon( int matrix_layout, char norm, lapack_int n, const float* a,
                            lapack_int lda, float anorm, float* rcond );
lapack_int LAPACKE_dgecon( int matrix_layout, char norm, lapack_int n, const double* a,
                            lapack_int lda, double anorm, double* rcond );
lapack_int LAPACKE_cgecon( int matrix_layout, char norm, lapack_int n,
                           const lapack_complex_float* a, lapack_int lda, float anorm, float* rcond );
lapack_int LAPACKE_zgecon( int matrix_layout, char norm, lapack_int n,
                           const lapack_complex_double* a, lapack_int lda, double anorm, double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a general matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Before calling this routine:

- compute anorm (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call [?getrf](#) to compute the LU factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i> , <i>work</i>	REAL for sgecon DOUBLE PRECISION for dgecon COMPLEX for cgecon DOUBLE COMPLEX for zgecon. Arrays: <i>a</i> (<i>lda</i> , *), <i>work</i> (*). The array <i>a</i> contains the LU -factored matrix A , as returned by ?getrf . The second dimension of <i>a</i> must be at least $\max(1, n)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 4*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>anorm</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The norm of the original matrix A (see Description).
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for cgecon DOUBLE PRECISION for zgecon. Workspace array, DIMENSION at least $\max(1, 2*n)$.

Output Parameters

<i>rcond</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gecon` interface are as follows:

`a` Holds the matrix A of size (n, n) .

`norm` Must be '`1`', '`0`', or '`I`'. The default value is '`1`'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$ or $A^{H*}x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2*n^2$ floating-point operations for real flavors and $8*n^2$ for complex flavors.

?gbcon

Estimates the reciprocal of the condition number of a band matrix in the 1-norm or the infinity-norm.

Syntax

FORTRAN 77:

```
call sgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info )
call dgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, iwork, info )
call cgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info )
call zgbcon( norm, n, kl, ku, ab, ldab, ipiv, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call gbcon( ab, ipiv, anorm, rcond [,kl] [,norm] [,info] )
```

C:

```
lapack_int LAPACKE_sgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl,
                           lapack_int ku, const float* ab, lapack_int ldab, const lapack_int* ipiv, float anorm,
                           float* rcond );
lapack_int LAPACKE_dgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl,
                           lapack_int ku, const double* ab, lapack_int ldab, const lapack_int* ipiv, double
                           anorm, double* rcond );
lapack_int LAPACKE_cgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl,
                           lapack_int ku, const lapack_complex_float* ab, lapack_int ldab, const lapack_int* ipiv,
                           float anorm, float* rcond );
lapack_int LAPACKE_zgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl,
                           lapack_int ku, const lapack_complex_double* ab, lapack_int ldab, const lapack_int*
                           ipiv, double anorm, double* rcond );
```

Include Files

- Fortran: `mkl.fi`

- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a general band matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = ||A||_1 ||A^{-1}||_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = ||A||_\infty ||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Before calling this routine:

- compute anorm (either $||A||_1 = \max_j \sum_i |a_{ij}|$ or $||A||_\infty = \max_i \sum_j |a_{ij}|$)
- call [?gbtrf](#) to compute the LU factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'.
	If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm.
	If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>k1</i>	INTEGER. The number of subdiagonals within the band of A ; $k1 \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> . ($ldab \geq 2*k1 + ku + 1$).
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?gbtrf .
<i>ab, work</i>	REAL for sgbccon DOUBLE PRECISION for dgbccon COMPLEX for cgbccon DOUBLE COMPLEX for zgbccon. Arrays: <i>ab</i> (<i>ldab, *</i>), <i>work</i> (*).
	The array <i>ab</i> contains the factored band matrix A , as returned by ?gbtrf .
	The second dimension of <i>ab</i> must be at least $\max(1, n)$. The array <i>work</i> is a workspace for the routine.
	The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>anorm</i>	REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see [Description](#)).

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork

REAL for cgbcon

DOUBLE PRECISION for zgbcon .

Workspace array, DIMENSION at least $\max(1, 2*n)$.

Output Parameters

rcond

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

INTEGER. If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine gbcon interface are as follows:

ab

Holds the array A of size $(2*k1+ku+1, n)$.

ipiv

Holds the vector of length n .

norm

Must be '1', 'O', or 'I'. The default value is '1'.

k1

If omitted, assumed $k1 = ku$.

ku

Restored as $ku = lda - 2*k1 - 1$.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$ or $A^{H*}x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n(ku + 2k1)$ floating-point operations for real flavors and $8n(ku + 2k1)$ for complex flavors.

?gtcon

Estimates the reciprocal of the condition number of a tridiagonal matrix using the factorization computed by [?gttrf](#).

Syntax

FORTRAN 77:

```
call sgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info )
call dgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, iwork, info )
call cgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
call zgtcon( norm, n, dl, d, du, du2, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call gtcon( dl, d, du, du2, ipiv, anorm, rcond [,norm] [,info] )
```

C:

```
lapack_int LAPACKE_sgtcon( char norm, lapack_int n, const float* dl, const float* d,
                           const float* du, const float* du2, const lapack_int* ipiv, float anorm, float* rcond );
lapack_int LAPACKE_dgtcon( char norm, lapack_int n, const double* dl, const double* d,
                           const double* du, const double* du2, const lapack_int* ipiv, double anorm, double* rcond );
lapack_int LAPACKE_cgtcon( char norm, lapack_int n, const lapack_complex_float* dl,
                           const lapack_complex_float* d, const lapack_complex_float* du,
                           const lapack_complex_float* du2, const lapack_int* ipiv, float anorm, float* rcond );
lapack_int LAPACKE_zgtcon( char norm, lapack_int n, const lapack_complex_double* dl,
                           const lapack_complex_double* d, const lapack_complex_double* du,
                           const lapack_complex_double* du2, const lapack_int* ipiv, double anorm, double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a real or complex tridiagonal matrix A in the 1-norm or infinity-norm:

$$\kappa_1(A) = ||A||_1 ||A^{-1}||_1$$

$$\kappa_\infty(A) = ||A||_\infty ||A^{-1}||_\infty$$

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as $rcond = 1 / (||A|| \cdot ||A^{-1}||)$.

Before calling this routine:

- compute $anorm$ (either $||A||_1 = \max_j \sum_i |a_{ij}|$ or $||A||_\infty = \max_i \sum_j |a_{ij}|$)
- call `?gttrf` to compute the LU factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

norm

CHARACTER*1. Must be '1' or 'O' or 'I'.

If $norm = '1'$ or ' ∞ ', then the routine estimates the condition number of matrix A in 1-norm.

If $norm = 'I'$, then the routine estimates the condition number of matrix A in infinity-norm.

n

INTEGER. The order of the matrix A ; $n \geq 0$.

$dl, d, du, du2$

REAL for sgtcon

DOUBLE PRECISION for dgtcon

COMPLEX for cgtcon

DOUBLE COMPLEX for zgtcon.

Arrays: $dl(n - 1)$, $d(n)$, $du(n - 1)$, $du2(n - 2)$.

The array dl contains the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A as computed by [?gttrf](#).

The array d contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A .

The array du contains the $(n - 1)$ elements of the first superdiagonal of U .

The array $du2$ contains the $(n - 2)$ elements of the second superdiagonal of U .

$ipiv$

INTEGER.

Array, DIMENSION (n) . The array of pivot indices, as returned by [?gttrf](#).

$anorm$

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

$work$

REAL for sgtcon

DOUBLE PRECISION for dgtcon

COMPLEX for cgtcon

DOUBLE COMPLEX for zgtcon.

Workspace array, DIMENSION $(2*n)$.

$iwork$

INTEGER. Workspace array, DIMENSION (n) . Used for real flavors only.

Output Parameters

$rcond$

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond=0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gtcon` interface are as follows:

<i>d1</i>	Holds the vector of length (<i>n</i> -1).
<i>d</i>	Holds the vector of length <i>n</i> .
<i>du</i>	Holds the vector of length (<i>n</i> -1).
<i>du2</i>	Holds the vector of length (<i>n</i> -2).
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>norm</i>	Must be '1', 'O', or 'I'. The default value is '1'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?pocon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```
call spocon( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call dpocon( uplo, n, a, lda, anorm, rcond, work, iwork, info )
call cpocon( uplo, n, a, lda, anorm, rcond, work, rwork, info )
call zpocon( uplo, n, a, lda, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call pocon( a, anorm, rcond [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_spocon( int matrix_layout, char uplo, lapack_int n, const float* a,
                           lapack_int lda, float anorm, float* rcond );
lapack_int LAPACKE_dpocon( int matrix_layout, char uplo, lapack_int n, const double* a,
                           lapack_int lda, double anorm, double* rcond );
lapack_int LAPACKE_cpocon( int matrix_layout, char uplo, lapack_int n,
                           const lapack_complex_float* a, lapack_int lda, float anorm, float* rcond );
```

```
lapack_int LAPACKE_zpocon( int matrix_layout, char uplo, lapack_int n,
const lapack_complex_double* a, lapack_int lda, double anorm, double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix A :

$$\kappa_1(A) = ||A||_1 \cdot ||A^{-1}||_1 \quad (\text{since } A \text{ is symmetric or Hermitian}, \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute anorm (either $||A||_1 = \max_j \sum_i |a_{ij}|$ or $||A||_\infty = \max_i \sum_j |a_{ij}|$)
- call [?potrf](#) to compute the Cholesky factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <i>uplo</i> = 'U', the upper triangle of A is stored.
	If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i> , <i>work</i>	REAL for <i>sopocon</i> DOUBLE PRECISION for <i>dpocon</i> COMPLEX for <i>cpocon</i> DOUBLE COMPLEX for <i>zpocon</i> . Arrays: <i>a</i> (<i>lda</i> , *), <i>work</i> (*). The array <i>a</i> contains the factored matrix A , as returned by ?potrf . The second dimension of <i>a</i> must be at least $\max(1, n)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $\text{lda} \geq \max(1, n)$.
<i>anorm</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. The norm of the <i>original</i> matrix A (see <i>Description</i>).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pocon` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?ppcon

Estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```

call sppcon( uplo, n, ap, anorm, rcond, work, iwork, info )
call dppcon( uplo, n, ap, anorm, rcond, work, iwork, info )
call cppcon( uplo, n, ap, anorm, rcond, work, rwork, info )
call zppcon( uplo, n, ap, anorm, rcond, work, rwork, info )

```

Fortran 95:

```
call ppcon( ap, anorm, rcond [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_sppcon( int matrix_layout, char uplo, lapack_int n, const float* ap,
float anorm, float* rcond );

lapack_int LAPACKE_dppcon( int matrix_layout, char uplo, lapack_int n, const double* ap,
double anorm, double* rcond );

lapack_int LAPACKE_cppcon( int matrix_layout, char uplo, lapack_int n,
const lapack_complex_float* ap, float anorm, float* rcond );

lapack_int LAPACKE_zppcon( int matrix_layout, char uplo, lapack_int n,
const lapack_complex_double* ap, double anorm, double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-definite matrix A :

$$\kappa_1(A) = ||A||_1 \ ||A^{-1}||_1 \ (\text{since } A \text{ is symmetric or Hermitian}, \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute anorm (either $||A||_1 = \max_j \sum_i |a_{ij}|$ or $||A||_\infty = \max_i \sum_j |a_{ij}|$)
- call [?ptrf](#) to compute the Cholesky factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>ap, work</i>	REAL for sppcon DOUBLE PRECISION for dppcon COMPLEX for cppcon DOUBLE COMPLEX for zppcon. Arrays: <i>ap</i> (*), <i>work</i> (*).
	The array <i>ap</i> contains the packed factored matrix A , as returned by ?ptrf . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>anorm</i>	REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

iwork INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork REAL for *cpbcon*

DOUBLE PRECISION for *zpbcon*.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

rcond REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets *rcond* = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime *rcond* is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *ppbcon* interface are as follows:

ap Holds the array A of size $(n * (n+1) / 2)$.

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?pbcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix.

Syntax

FORTRAN 77:

```
call spbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info )
call dpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, iwork, info )
call cpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info )
call zpbcon( uplo, n, kd, ab, ldab, anorm, rcond, work, rwork, info )
```

Fortran 95:

```
call pbcon( ab, anorm, rcond [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_spbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
                           const float* ab, lapack_int ldab, float anorm, float* rcond );
lapack_int LAPACKE_dpbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
                           const double* ab, lapack_int ldab, double anorm, double* rcond );
lapack_int LAPACKE_cpbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
                           const lapack_complex_float* ab, lapack_int ldab, float anorm, float* rcond );
lapack_int LAPACKE_zpbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
                           const lapack_complex_double* ab, lapack_int ldab, double anorm, double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix A :

$$\kappa_1(A) = ||A||_1 \cdot ||A^{-1}||_1 \quad (\text{since } A \text{ is symmetric or Hermitian}, \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute anorm (either $||A||_1 = \max_j \sum_i |a_{ij}|$ or $||A||_\infty = \max_i \sum_j |a_{ij}|$)
- call [?pbtrf](#) to compute the Cholesky factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <i>uplo</i> = 'U', the upper triangular factor is stored in <i>ab</i> .
	If <i>uplo</i> = 'L', the lower triangular factor is stored in <i>ab</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> . ($ldab \geq kd + 1$).
<i>ab, work</i>	REAL for spbcon DOUBLE PRECISION for dpbcon COMPLEX for cpbcon DOUBLE COMPLEX for zpbcon.

Arrays: $ab(1:dab, \cdot)$, $work(\cdot)$.

The array *ab* contains the factored matrix *A* in band form, as returned by [?pbtrf](#). The second dimension of *ab* must be at least $\max(1, n)$.

The array `work` is a workspace for the routine. The dimension of `work` must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

anorm

`REAL` for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork

REAL for cpbcon

DOUBLE PRECISION for zpbcon.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

INTEGER. If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbcon` interface are as follows:

ab

Holds the array A of size $(kd+1, n)$.

uplo

Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4*n(kd + 1)$ floating-point operations for real flavors and $16*n(kd + 1)$ for complex flavors.

?ptcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax**FORTRAN 77:**

```
call sptcon( n, d, e, anorm, rcond, work, info )
call dptcon( n, d, e, anorm, rcond, work, info )
call cptcon( n, d, e, anorm, rcond, work, info )
call zptcon( n, d, e, anorm, rcond, work, info )
```

Fortran 95:

```
call ptcon( d, e, anorm, rcond [,info] )
```

C:

```
lapack_int LAPACKE_sptcon( lapack_int n, const float* d, const float* e, float anorm,
float* rcond );
lapack_int LAPACKE_dptcon( lapack_int n, const double* d, const double* e, double
anorm, double* rcond );
lapack_int LAPACKE_cptcon( lapack_int n, const float* d, const lapack_complex_float* e,
float anorm, float* rcond );
lapack_int LAPACKE_zptcon( lapack_int n, const double* d, const lapack_complex_double*
e, double anorm, double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the reciprocal of the condition number (in the 1-norm) of a real symmetric or complex Hermitian positive-definite tridiagonal matrix using the factorization $A = L*D*L^T$ for real flavors and $A = L*D*L^H$ for complex flavors or $A = U^T*D*U$ for real flavors and $A = U^H*D*U$ for complex flavors computed by [?pttrf](#) :

$$\kappa_1(A) = ||A||_1 \cdot ||A^{-1}||_1 \quad (\text{since } A \text{ is symmetric or Hermitian}, \kappa_\infty(A) = \kappa_1(A)).$$

The norm $||A^{-1}||$ is computed by a direct method, and the reciprocal of the condition number is computed as $rcond = 1 / (||A|| \cdot ||A^{-1}||)$.

Before calling this routine:

- compute *anorm* as $||A||_1 = \max_j \sum_i |a_{ij}|$
- call [?pttrf](#) to compute the factorization of *A*.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>d, work</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, dimension (n). The array d contains the n diagonal elements of the diagonal matrix D from the factorization of A , as computed by ?pttrf ; $work$ is a workspace array.
<i>e</i>	REAL for $sptcon$ DOUBLE PRECISION for $dptcon$ COMPLEX for $cptcon$ DOUBLE COMPLEX for $zptcon$. Array, DIMENSION ($n - 1$). Contains off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by ?pttrf .
<i>anorm</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The 1- norm of the <i>original</i> matrix A (see <i>Description</i>).

Output Parameters

<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gtcon` interface are as follows:

<i>d</i>	Holds the vector of length n .
<i>e</i>	Holds the vector of length ($n-1$).

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4*n(kd + 1)$ floating-point operations for real flavors and $16*n(kd + 1)$ for complex flavors.

?sycon

Estimates the reciprocal of the condition number of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssycon( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call dsycon( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call csycon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zsycon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call sycon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_ssycon( int matrix_layout, char uplo, lapack_int n, const float* a,
                            lapack_int lda, const lapack_int* ipiv, float anorm, float* rcond );
lapack_int LAPACKE_dsycon( int matrix_layout, char uplo, lapack_int n, const double* a,
                            lapack_int lda, const lapack_int* ipiv, double anorm, double* rcond );
lapack_int LAPACKE_csycon( int matrix_layout, char uplo, lapack_int n,
                           const lapack_complex_float* a, lapack_int lda, const lapack_int* ipiv, float anorm,
                           float* rcond );
lapack_int LAPACKE_zsycon( int matrix_layout, char uplo, lapack_int n,
                           const lapack_complex_double* a, lapack_int lda, const lapack_int* ipiv, double anorm,
                           double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a symmetric matrix A :

$$\kappa_1(A) = ||A||_1 \cdot ||A^{-1}||_1 \quad (\text{since } A \text{ is symmetric}, \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute $anorm$ (either $||A||_1 = \max_j \sum_i |a_{ij}|$ or $||A||_\infty = \max_i \sum_j |a_{ij}|$)
- call [?sytrf](#) to compute the factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = P^*U*D^*U^T*P^T$.
	If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = P^*L*D^*L^T*P^T$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>a, work</i>	<p>REAL for ssycon DOUBLE PRECISION for dsycon COMPLEX for csycon DOUBLE COMPLEX for zsycon.</p> <p>Arrays: $a(1:\text{lda}, :)$, $work(1:\text{max}(1, 2*n))$.</p> <p>The array a contains the factored matrix A, as returned by ?sytrf. The second dimension of a must be at least $\max(1, n)$.</p> <p>The array $work$ is a workspace for the routine. The dimension of $work$ must be at least $\max(1, 2*n)$.</p>
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. <p>The array $ipiv$, as returned by ?sytrf.</p>
<i>anorm</i>	<p>REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.</p> <p>The norm of the <i>original</i> matrix A (see <i>Description</i>).</p>
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>rcond</i>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.</p>
<i>info</i>	<p>INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i-th parameter had an illegal value.</p>

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sycon` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?sycon_rook

Estimates the reciprocal of the condition number of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call dsycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, iwork, info )
call csycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zsycon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call sycon_rook( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine estimates the reciprocal of the condition number of a symmetric matrix A :

$$\kappa_1(A) = ||A||_1 \cdot ||A^{-1}||_1 \quad (\text{since } A \text{ is symmetric}, \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute $anorm$ (either $||A||_1 = \max_j \sum_i |a_{ij}|$ or $||A||_\infty = \max_i \sum_j |a_{ij}|$)
- call `?sytrf_rook` to compute the factorization of A .

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '.
-------------------	--

Indicates how the input matrix A has been factored:

If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = U*D*U^T$.

If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = L*D*L^T$.

n

INTEGER. The order of matrix A ; $n \geq 0$.

$a, work$

REAL for ssycon_rook

DOUBLE PRECISION for dsycon_rook

COMPLEX for csycon_rook

DOUBLE COMPLEX for zsycon_rook.

Arrays: $a(1:n, :)$, $work(*)$.

The array a contains the factored matrix A , as returned by [?sytrf_rook](#). The second dimension of a must be at least $\max(1, n)$.

The array $work$ is a workspace for the routine.

The dimension of $work$ must be at least $\max(1, 2*n)$.

lda

INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

$ipiv$

INTEGER. Array, size at least $\max(1, n)$.

The array $ipiv$, as returned by [?sytrf_rook](#).

$anorm$

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

$iwork$

INTEGER. Workspace array, size at least $\max(1, n)$.

Output Parameters

$rcond$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

$info$

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sycon_rook` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?hecon

Estimates the reciprocal of the condition number of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call checon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zhecon( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call hecon( a, ipiv, anorm, rcond [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_checon( int matrix_layout, char uplo, lapack_int n,
const lapack_complex_float* a, lapack_int lda, const lapack_int* ipiv, float anorm,
float* rcond );
lapack_int LAPACKE_zhecon( int matrix_layout, char uplo, lapack_int n,
const lapack_complex_double* a, lapack_int lda, const lapack_int* ipiv, double anorm,
double* rcond );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix A :

$$\kappa_1(A) = ||A||_1 \cdot ||A^{-1}||_1 \quad (\text{since } A \text{ is Hermitian}, \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute $anorm$ (either $||A||_1 = \max_j \sum_i |a_{ij}|$ or $||A||_\infty = \max_i \sum_j |a_{ij}|$)
- call `?hetrf` to compute the factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix A has been factored:

If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = P^*U*D*U^H*P^T$.

If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = P^*L*D*L^H*P^T$.

*n*INTEGER. The order of matrix A ; $n \geq 0$.*a, work*

COMPLEX for checon

DOUBLE COMPLEX for zhecon.

Arrays: $a(1da, *)$, $work(*)$.

The array a contains the factored matrix A , as returned by [?hetrf](#).
The second dimension of a must be at least $\max(1, n)$.

The array $work$ is a workspace for the routine.

The dimension of $work$ must be at least $\max(1, 2*n)$.

*lda*INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.*ipiv*INTEGER. Array, DIMENSION at least $\max(1, n)$.

The array $ipiv$, as returned by [?hetrf](#).

anorm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

Output Parameters

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

*info*INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hecon` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

?hecon_rook

Estimates the reciprocal of the condition number of a Hermitian matrix using factorization obtained with one of the bounded diagonal pivoting methods (max 2 interchanges).

Syntax

FORTRAN 77:

```
call checon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
call zhecon_rook( uplo, n, a, lda, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call hecon_rook( a, ipiv, anorm, rcond [,uplo] [,info] )
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix A using the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ computed by [?hetrf_rook](#).

An estimate is obtained for $\text{norm}(A^{-1})$, and the reciprocal of the condition number is computed as $rcond = 1/(anorm*\text{norm}(A^{-1}))$.

Input Parameters

`uplo` CHARACTER*1. Must be '`U`' or '`L`'.

Indicates how the input matrix A has been factored:

If `uplo` = '`U`', the array `a` stores the upper triangular factor U of the factorization $A = U^*D^*U^H$.

If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = L \cdot D \cdot L^H$.

n

INTEGER. The order of matrix A ; $n \geq 0$.

a, work

COMPLEX for checon_rook

COMPLEX*16 for zhecon_rook.

Arrays: $a(1:n, 1:n)$, $work(1:n)$.

The array a contains the factored matrix A , as returned by [?hetrf_rook](#). The second dimension of a must be at least $\max(1, n)$.

The array $work$ is a workspace for the routine. The dimension of $work$ must be at least $\max(1, 2*n)$.

lda

INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ipiv

INTEGER. Array, size at least $\max(1, n)$.

The array $ipiv$, as returned by [?hetrf_rook](#).

anorm

REAL for checon_rook

DOUBLE PRECISION for zhecon_rook.

The 1-norm of the original matrix A (see *Description*).

Output Parameters

rcond

REAL for checon_rook

DOUBLE PRECISION for zhecon_rook.

The reciprocal of the condition number of the matrix A , computed as $rcond = 1/(anorm * ainvnm)$, where $ainvnm$ is an estimate of the 1-norm of A^{-1} computed in this routine.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hecon_rook` interface are as follows:

a

Holds the matrix A of size (n, n) .

ipiv

Holds the vector of length n .

uplo

Must be '`U`' or '`L`'. The default value is '`U`'.

See Also

[Matrix Storage Schemes](#)

?spcon

Estimates the reciprocal of the condition number of a packed symmetric matrix.

Syntax**FORTRAN 77:**

```
call sspcon( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call dspcon( uplo, n, ap, ipiv, anorm, rcond, work, iwork, info )
call cspcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call zspcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call spcon( ap, ipiv, anorm, rcond [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_sspcon( int matrix_layout, char uplo, lapack_int n, const float* ap,
const lapack_int* ipiv, float anorm, float* rcond );
lapack_int LAPACKE_dspcon( int matrix_layout, char uplo, lapack_int n, const double* ap,
const lapack_int* ipiv, double anorm, double* rcond );
lapack_int LAPACKE_cspcon( int matrix_layout, char uplo, lapack_int n,
const lapack_complex_float* ap, const lapack_int* ipiv, float anorm, float* rcond );
lapack_int LAPACKE_zspcon( int matrix_layout, char uplo, lapack_int n,
const lapack_complex_double* ap, const lapack_int* ipiv, double anorm, double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a packed symmetric matrix A :

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 \text{ (since } A \text{ is symmetric, } \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute anorm (either $\|A\|_1 = \max_j \sum_i |a_{ij}|$ or $\|A\|_\infty = \max_i \sum_j |a_{ij}|$)
- call `?sptrf` to compute the factorization of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix A has been factored:

If $uplo = 'U'$, the array ap stores the packed upper triangular factor U of the factorization $A = P^*U^*D^*U^T*P^T$.

If $uplo = 'L'$, the array ap stores the packed lower triangular factor L of the factorization $A = P^T L D L^T P$.

n

INTEGER. The order of matrix A ; $n \geq 0$.

ap, work

REAL for sspcon

DOUBLE PRECISION for dspcon

COMPLEX for cspcon

DOUBLE COMPLEX for zspcon.

Arrays: $ap(*)$, $work(*)$.

The array ap contains the packed factored matrix A , as returned by [?sptrf](#). The dimension of ap must be at least $\max(1, n(n+1)/2)$.

The array $work$ is a workspace for the routine.

The dimension of $work$ must be at least $\max(1, 2*n)$.

ipiv

INTEGER. Array, DIMENSION at least $\max(1, n)$.

The array $ipiv$, as returned by [?sptrf](#).

anorm

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

rcond

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spcon` interface are as follows:

ap

Holds the array A of size $(n*(n+1)/2)$.

ipiv

Holds the vector of length n .

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.

?hpcon

Estimates the reciprocal of the condition number of a packed Hermitian matrix.

Syntax

FORTRAN 77:

```
call chpcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
call zhpcon( uplo, n, ap, ipiv, anorm, rcond, work, info )
```

Fortran 95:

```
call hpcon( ap, ipiv, anorm, rcond [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_chpcon( int matrix_layout, char uplo, lapack_int n,
                           const lapack_complex_float* ap, const lapack_int* ipiv, float anorm, float* rcond );
lapack_int LAPACKE_zhpcon( int matrix_layout, char uplo, lapack_int n,
                           const lapack_complex_double* ap, const lapack_int* ipiv, double anorm, double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix *A*:

$$\kappa_1(A) = ||A||_1 \cdot ||A^{-1}||_1 \quad (\text{since } A \text{ is Hermitian}, \kappa_\infty(A) = \kappa_1(A)).$$

Before calling this routine:

- compute *anorm* (either $||A||_1 = \max_j \sum_i |a_{ij}|$ or $||A||_\infty = \max_i \sum_j |a_{ij}|$)
- call ?hptrf to compute the factorization of *A*.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix *A* has been factored:

If *uplo* = 'U', the array *ap* stores the packed upper triangular factor *U* of the factorization $A = P^*U^*D^*U^T*P^T$.

If $uplo = 'L'$, the array ap stores the packed lower triangular factor L of the factorization $A = P^T L D L^T P$.

n INTEGER. The order of matrix A ; $n \geq 0$.

$ap, work$ COMPLEX for chpcon

DOUBLE COMPLEX for zhpcon.

Arrays: $ap(*)$, $work(*)$.

The array ap contains the packed factored matrix A , as returned by [?hptrf](#). The dimension of ap must be at least $\max(1, n(n+1)/2)$.

The array $work$ is a workspace for the routine.

The dimension of $work$ must be at least $\max(1, 2*n)$.

$ipiv$ INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array $ipiv$, as returned by [?hptrf](#).

$anorm$ REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

The norm of the *original* matrix A (see *Description*).

Output Parameters

$rcond$ REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

$info$ INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbcon` interface are as follows:

ap Holds the array A of size $(n*(n+1)/2)$.

$ipiv$ Holds the vector of length n .

Application Notes

The computed $rcond$ is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

?trcon

Estimates the reciprocal of the condition number of a triangular matrix.

Syntax

FORTRAN 77:

```
call strcon( norm, uplo, diag, n, a, lda, rcond, work, iwork, info )
call dtrcon( norm, uplo, diag, n, a, lda, rcond, work, iwork, info )
call ctrcon( norm, uplo, diag, n, a, lda, rcond, work, rwork, info )
call ztrcon( norm, uplo, diag, n, a, lda, rcond, work, rwork, info )
```

Fortran 95:

```
call trcon( a, rcond [,uplo] [,diag] [,norm] [,info] )
```

C:

```
lapack_int LAPACKE_strcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, const float* a, lapack_int lda, float* rcond );
lapack_int LAPACKE_dtrcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, const double* a, lapack_int lda, double* rcond );
lapack_int LAPACKE_ctrcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, const lapack_complex_float* a, lapack_int lda, float* rcond );
lapack_int LAPACKE_ztrcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, const lapack_complex_double* a, lapack_int lda, double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a triangular matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

norm

CHARACTER*1. Must be '1' or 'O' or 'I'.

If $norm = '1'$ or ' ∞ ', then the routine estimates the condition number of matrix A in 1-norm.

If $norm = 'I'$, then the routine estimates the condition number of matrix A in infinity-norm.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether A is upper or lower triangular:

If $uplo = 'U'$, the array a stores the upper triangle of A , other array elements are not referenced.

If $uplo = 'L'$, the array a stores the lower triangle of A , other array elements are not referenced.

diag

CHARACTER*1. Must be 'N' or 'U'.

If $diag = 'N'$, then A is not a unit triangular matrix.

If $diag = 'U'$, then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array a .

n

INTEGER. The order of the matrix A ; $n \geq 0$.

a, work

REAL for strcon

DOUBLE PRECISION for dtrcon

COMPLEX for ctrcon

DOUBLE COMPLEX for ztrcon.

Arrays: $a(1:n, :)$, $work(1:n)$.

The array a contains the matrix A . The second dimension of a must be at least $\max(1, n)$.

The array $work$ is a workspace for the routine. The dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

lda

INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork

REAL for ctrcon

DOUBLE PRECISION for ztrcon.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

rcond

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trcon` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>norm</i>	Must be '1', 'O', or 'I'. The default value is '1'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than 10*r*. A call to this routine involves solving a number of systems of linear equations *A***x* = *b*; the number is usually 4 or 5 and never more than 11. Each solution requires approximately *n*² floating-point operations for real flavors and 4*n*² operations for complex flavors.

?tpcon

Estimates the reciprocal of the condition number of a packed triangular matrix.

Syntax

FORTRAN 77:

```
call stpcon( norm, uplo, diag, n, ap, rcond, work, iwork, info )
call dtpcon( norm, uplo, diag, n, ap, rcond, work, iwork, info )
call ctpcon( norm, uplo, diag, n, ap, rcond, work, rwork, info )
call ztpcon( norm, uplo, diag, n, ap, rcond, work, rwork, info )
```

Fortran 95:

```
call tpcon( ap, rcond [,uplo] [,diag] [,norm] [,info] )
```

C:

```
lapack_int LAPACKE_stpcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, const float* ap, float* rcond );
lapack_int LAPACKE_dtpcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, const double* ap, double* rcond );
lapack_int LAPACKE_ctpcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, const lapack_complex_float* ap, float* rcond );
lapack_int LAPACKE_ztpcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, const lapack_complex_double* ap, double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a packed triangular matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangle of A in packed form. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangle of A in packed form.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>ap, work</i>	REAL for stpcon DOUBLE PRECISION for dtpcon COMPLEX for ctpcon DOUBLE COMPLEX for ztpcon. Arrays: <i>ap(*)</i> , <i>work(*)</i> . The array <i>ap</i> contains the packed matrix A . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>work</i> is a workspace for the routine. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>ctpcon</i> DOUBLE PRECISION for <i>ztpcon</i> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>rcond</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.
	An estimate of the reciprocal of the condition number. The routine sets <i>rcond</i> = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime <i>rcond</i> is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *tpcon* interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>norm</i>	Must be '1', 'O', or 'I'. The default value is '1'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and $4n^2$ operations for complex flavors.

?tbcon

Estimates the reciprocal of the condition number of a triangular band matrix.

Syntax

FORTRAN 77:

```
call stbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call dtbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, iwork, info )
call ctbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
call ztbcon( norm, uplo, diag, n, kd, ab, ldab, rcond, work, rwork, info )
```

Fortran 95:

```
call tbcon( ab, rcond [,uplo] [,diag] [,norm] [,info] )
```

C:

```
lapack_int LAPACKE_stbcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, lapack_int kd, const float* ab, lapack_int ldab, float* rcond );
lapack_int LAPACKE_dtbcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, lapack_int kd, const double* ab, lapack_int ldab, double* rcond );
lapack_int LAPACKE_ctbcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, lapack_int kd, const lapack_complex_float* ab, lapack_int ldab, float* rcond );
lapack_int LAPACKE_ztbcon( int matrix_layout, char norm, char uplo, char diag,
                           lapack_int n, lapack_int kd, const lapack_complex_double* ab, lapack_int ldab, double* rcond );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the reciprocal of the condition number of a triangular band matrix A in either the 1-norm or infinity-norm:

$$\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)$$

$$\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).$$

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>norm</i>	CHARACTER*1. Must be '1' or 'O' or 'I'. If <i>norm</i> = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm. If <i>norm</i> = 'I', then the routine estimates the condition number of matrix A in infinity-norm.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangle of A in packed form. If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangle of A in packed form.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix.

If $diag = 'U'$, then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array ab .

n

INTEGER. The order of the matrix A ; $n \geq 0$.

kd

INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.

ab, work

REAL for `stbcon`

DOUBLE PRECISION for `dtbcon`

COMPLEX for `ctbcon`

DOUBLE COMPLEX for `ztbcon`.

Arrays: $ab(1:dab, *)$, $work(*)$.

The array ab contains the band matrix A . The second dimension of ab must be at least $\max(1, n)$. The array $work$ is a workspace for the routine.

The dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldab

INTEGER. The leading dimension of the array ab . ($ldab \geq kd + 1$).

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork

REAL for `ctbcon`

DOUBLE PRECISION for `ztbcon`.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

rcond

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tbccon` interface are as follows:

ab

Holds the array A of size $(kd+1, n)$.

<i>norm</i>	Must be '1', 'O', or 'I'. The default value is '1'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The computed *rcond* is never less than *r* (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2*n(kd + 1)$ floating-point operations for real flavors and $8*n(kd + 1)$ operations for complex flavors.

Refining the Solution and Estimating Its Error

This section describes the LAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see [Routines for Matrix Factorization](#) and [Routines for Solving Systems of Linear Equations](#)).

?gerfs

Refines the solution of a system of linear equations with a general matrix and estimates its error.

Syntax

FORTRAN 77:

```
call sgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
iwork, info )
call dgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
iwork, info )
call cgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
rwork, info )
call zgerfs( trans, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work,
rwork, info )
```

Fortran 95:

```
call gerfs( a, af, ipiv, b, x [,trans] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_sgerfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf,
const lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx,
float* ferr, float* berr );
lapack_int LAPACKE_dgerfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf,
const lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx,
double* ferr, double* berr );
lapack_int LAPACKE_cgerfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af,
lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_zgerfs( int matrix_layout, char trans, lapack_int n, lapack_int
nrhs, const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af,
lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int
ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^T X = B$ or $A^H X = B$ with a general matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?getrf](#)
- call the solver routine [?getrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A^T X = B$. If <i>trans</i> = 'T', the system has the form $A^H X = B$. If <i>trans</i> = 'C', the system has the form $A^T X = B$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a,af,b,x,work</i>	REAL for sgerfs DOUBLE PRECISION for dgerfs COMPLEX for cgerfs DOUBLE COMPLEX for zgerfs. Arrays: <i>a</i> (<i>lda</i> ,*) contains the original matrix A , as supplied to ?getrf . <i>af</i> (<i>ldaf</i> ,*) contains the factored matrix A , as returned by ?getrf . <i>b</i> (<i>ldb</i> ,*) contains the right-hand side matrix B .

`x(1:dx, *)` contains the solution matrix X .

`work(*)` is a workspace array.

The second dimension of `a` and `af` must be at least $\max(1, n)$; the second dimension of `b` and `x` must be at least $\max(1, nrhs)$; the dimension of `work` must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

<code>lda</code>	INTEGER. The leading dimension of <code>a</code> ; $lda \geq \max(1, n)$.
<code>ldaf</code>	INTEGER. The leading dimension of <code>af</code> ; $ldaf \geq \max(1, n)$.
<code>ldb</code>	INTEGER. The leading dimension of <code>b</code> ; $ldb \geq \max(1, n)$.
<code>ldx</code>	INTEGER. The leading dimension of <code>x</code> ; $ldx \geq \max(1, n)$.
<code>ipiv</code>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <code>ipiv</code> array, as returned by ?getrf .
<code>iwork</code>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<code>rwork</code>	REAL for <code>cgerfs</code> DOUBLE PRECISION for <code>zgerfs</code> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<code>x</code>	The refined solution matrix X .
<code>ferr, berr</code>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gerfs` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>af</code>	Holds the matrix AF of size (n, n) .

<i>ipiv</i>	Holds the vector of length n .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?gerfsx

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a general matrix A and provides error bounds and backward error estimates.

Syntax

FORTRAN 77:

```
call sgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, idx,
rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork,
info )
call dgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, idx,
rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork,
info )
call cgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, idx,
rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork,
info )
call zgerfsx( trans, equed, n, nrhs, a, lda, af, ldaf, ipiv, r, c, b, ldb, x, idx,
rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork,
info )
```

C:

```
lapack_int LAPACKE_sgerfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf,
const lapack_int* ipiv, const float* r, const float* c, const float* b, lapack_int ldb,
float* x, lapack_int idx, float* rcond, float* berr, lapack_int n_err_bnds,
float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );
```

```

lapack_int LAPACKE_dgerfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf,
const lapack_int* ipiv, const double* r, const double* c, const double* b, lapack_int ldb,
double* x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds,
double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );

lapack_int LAPACKE_cgerfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda,
const lapack_complex_float* af, lapack_int ldaf, const lapack_int* ipiv, const float* r,
const float* c, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float*
err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACKE_zgerfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda,
const lapack_complex_double* af, lapack_int ldaf, const lapack_int* ipiv, const double* r,
const double* c, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds,
double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters `equed`, `r`, and `c` below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>trans</code>	CHARACTER*1. Must be 'N', 'T', or 'C'. Specifies the form of the system of equations: If <code>trans</code> = 'N', the system has the form $A^*X = B$ (No transpose). If <code>trans</code> = 'T', the system has the form $A^{**}T^*X = B$ (Transpose). If <code>trans</code> = 'C', the system has the form $A^{***}H^*X = B$ (Conjugate transpose = Transpose).
<code>equed</code>	CHARACTER*1. Must be 'N', 'R', 'C', or 'B'. Specifies the form of equilibration that was done to A before calling this routine.

If $\text{equed} = \text{'N'}$, no equilibration was done.

If $\text{equed} = \text{'R'}$, row equilibration was done, that is, A has been premultiplied by $\text{diag}(r)$.

If $\text{equed} = \text{'C'}$, column equilibration was done, that is, A has been postmultiplied by $\text{diag}(c)$.

If $\text{equed} = \text{'B'}$, both row and column equilibration was done, that is, A has been replaced by $\text{diag}(r) * A * \text{diag}(c)$. The right-hand side B has been changed accordingly.

n

INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.

a, af, b, work

REAL for sgerfsx

DOUBLE PRECISION for dgerfsx

COMPLEX for cgerfsx

DOUBLE COMPLEX for zgerfsx.

Arrays: $a(1da, *)$, $af(1daf, *)$, $b(1db, *)$, $work(*)$.

The array a contains the original n -by- n matrix A .

The array af contains the factored form of the matrix A , that is, the factors L and U from the factorization $A = P * L * U$ as computed by [?getrf](#).

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 4 * n)$ for real flavors, and at least $\max(1, 2 * n)$ for complex flavors.

lda

INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldaf

INTEGER. The leading dimension of af ; $ldaf \geq \max(1, n)$.

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, n)$. Contains the pivot indices as computed by [?getrf](#); for row $1 \leq i \leq n$, row i of the matrix was interchanged with row $ipiv(i)$.

r, c

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: $r(n)$, $c(n)$. The array r contains the row scale factors for A , and the array c contains the column scale factors for A .

$\text{equed} = \text{'R'}$ or 'B' , A is multiplied on the left by $\text{diag}(r)$; if $\text{equed} = \text{'N'}$ or 'C' , r is not accessed.

If $\text{equed} = \text{'R'}$ or 'B' , each element of r must be positive.

If $\text{equed} = \text{'C'}$ or 'B' , A is multiplied on the right by $\text{diag}(c)$; if $\text{equed} = \text{'N'}$ or 'R' , c is not accessed.

If $\text{equed} = \text{'C'}$ or 'B' , each element of c must be positive.

Each element of r or c should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

x

REAL for sgerfsx

DOUBLE PRECISION for dgerfsx

COMPLEX for cgerfsx

DOUBLE COMPLEX for zgerfsx .

Array, DIMENSION ($l_{\text{dx}}, *$).

The solution matrix X as computed by [?getrs](#)

l_{dx}

INTEGER. The leading dimension of the output array x ; $l_{\text{dx}} \geq \max(1, n)$.

$n_{\text{err_bnds}}$

INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See err_bnds_norm and err_bnds_comp descriptions in *Output Arguments* section below.

n_{params}

INTEGER. Specifies the number of parameters set in params . If ≤ 0 , the params array is never referenced and default values are used.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION n_{params} . Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to n_{params} are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass $n_{\text{params}} = 0$, which prevents the source code from accessing the params argument.

$\text{params}(\text{la_linrx_itref_i} = 1)$: Whether to perform iterative refinement or not. Default: 1.0

=0.0 No refinement is performed and no error bounds are computed.

=1.0 Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support DOUBLE PRECISION.

(Other values are reserved for future use.)

$\text{params}(\text{la_linrx_ithresh_i} = 2)$: Maximum number of residual computations allowed for refinement.

Default

10

Aggressive Set to 100 to permit convergence using approximate factorizations or factorizations other than *LU*. If the factorization uses a technique other than Gaussian elimination, the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

`params(la_linrx_cwise_i = 3)` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.

rwork REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION at least $\max(1, 3*n)$; used in complex flavors only.

Output Parameters

x REAL for `sgerfsx`

DOUBLE PRECISION for `dgerfsx`

COMPLEX for `cgerfsx`

DOUBLE COMPLEX for `zgerfsx`.

The improved solution matrix *X*.

rcond REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector *x(j)*, that is, the smallest relative change in any element of *A* or *B* that makes *x(j)* an exact solution.

err_bnds_norm REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (*nrhs, n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_norm(:, err)` contains the following three fields:

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

`err=2`

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix Z .

Let $z = s^*a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (`nrhs, n_err_bnds`). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params}(3) = 0.0$), then err_bnds_comp is not accessed. If $n_{\text{err_bnds}} < 3$, then at most the first $(:, n_{\text{err_bnds}})$ entries are returned.

The first index in $\text{err_bnds_comp}(i, :)$ corresponds to the i -th right-hand side.

The second index in $\text{err_bnds_comp}(:, \text{err})$ contains the following three fields:

$\text{err}=1$ "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

$\text{err}=2$ "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

$\text{err}=3$ Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Output parameter only if the input contains erroneous values, namely, in $\text{params}(1)$, $\text{params}(2)$, $\text{params}(3)$. In such a case, the corresponding elements of params are filled with default values on output.

info

INTEGER. If $\text{info} = 0$, the execution is successful. The solution to every right-hand side is guaranteed.

If $\text{info} = -i$, the i -th parameter had an illegal value.

If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $\text{rcond} = 0$ is returned.

If $\text{info} = n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $\text{params}(3) = 0.0$, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that $\text{err_bnds_norm}(j, 1) = 0.0$ or $\text{err_bnds_comp}(j, 1) = 0.0$). See the definition of $\text{err_bnds_norm}(:, 1)$ and $\text{err_bnds_comp}(:, 1)$. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp .

?gbrfs

Refines the solution of a system of linear equations with a general band matrix and estimates its error.

Syntax

FORTRAN 77:

```
call sgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )
call dgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr,
berr, work, iwork, info )
call cgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
call zgbrfs( trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, b, ldb, x, ldx, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call gbrfs( ab, afb, ipiv, b, x [,kl] [,trans] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_sgbrfs( int matrix_layout, char trans, lapack_int n, lapack_int kl,
lapack_int ku, lapack_int nrhs, const float* ab, lapack_int ldab, const float* afb,
lapack_int ldafb, const lapack_int* ipiv, const float* b, lapack_int ldb, float* x,
lapack_int ldx, float* ferr, float* berr );
lapack_int LAPACKE_dgbrfs( int matrix_layout, char trans, lapack_int n, lapack_int kl,
lapack_int ku, lapack_int nrhs, const double* ab, lapack_int ldab, const double* afb,
lapack_int ldafb, const lapack_int* ipiv, const double* b, lapack_int ldb, double* x,
lapack_int ldx, double* ferr, double* berr );
lapack_int LAPACKE_cgbrfs( int matrix_layout, char trans, lapack_int n, lapack_int kl,
lapack_int ku, lapack_int nrhs, const lapack_complex_float* ab, lapack_int ldab,
const lapack_complex_float* afb, lapack_int ldafb, const lapack_int* ipiv,
const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* ferr, float* berr );
```

```
lapack_int LAPACKE_zgbrfs( int matrix_layout, char trans, lapack_int n, lapack_int kl,
lapack_int ku, lapack_int nrhs, const lapack_complex_double* ab, lapack_int ldab,
const lapack_complex_double* afb, lapack_int ldafb, const lapack_int* ipiv,
const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int
ldx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?gbtrf](#)
- call the solver routine [?gbtrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A^*X = B$. If <i>trans</i> = 'T', the system has the form $A^T*X = B$. If <i>trans</i> = 'C', the system has the form $A^H*X = B$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of A ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of super-diagonals within the band of A ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab,afb,b,x,work</i>	REAL for sgbrfs DOUBLE PRECISION for dgbrfs COMPLEX for cgbrfs DOUBLE COMPLEX for zgbrfs. Arrays:

ab(*ldab*, *) contains the original band matrix *A*, as supplied to [?gbtrf](#), but stored in rows from 1 to *kl* + *ku* + 1.

afb(*ldafb*, *) contains the factored band matrix *A*, as returned by [?gbtrf](#).

b(*ldb*, *) contains the right-hand side matrix *B*.

x(*ldx*, *) contains the solution matrix *X*.

work(*) is a workspace array.

The second dimension of *ab* and *afb* must be at least $\max(1, n)$; the second dimension of *b* and *x* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldab

INTEGER. The leading dimension of *ab*.

ldafb

INTEGER. The leading dimension of *afb*.

ldb

INTEGER. The leading dimension of *b*; *ldb* $\geq \max(1, n)$.

ldx

INTEGER. The leading dimension of *x*; *ldx* $\geq \max(1, n)$.

ipiv

Array, DIMENSION at least $\max(1, n)$. The *ipiv* array, as returned by [?gbtrf](#).

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork

REAL for *cgbtrfs*

DOUBLE PRECISION for *zgbtrfs*.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

x

The refined solution matrix *X*.

ferr, *berr*

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gbrfs* interface are as follows:

<i>ab</i>	Holds the array A of size $(kl+ku+1, n)$.
<i>afb</i>	Holds the array AF of size $(2*kl*ku+1, n)$.
<i>ipiv</i>	Holds the vector of length n .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>kl</i>	If omitted, assumed $kl = ku$.
<i>ku</i>	Restored as $ku = lda - kl - 1$.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n(kl + ku)$ floating-point operations (for real flavors) or $16n(kl + ku)$ operations (for complex flavors). In addition, each step of iterative refinement involves $2n(4kl + 3ku)$ operations (for real flavors) or $8n(4kl + 3ku)$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?gbrfsx

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a banded matrix A and provides error bounds and backward error estimates.

Syntax

FORTRAN 77:

```
call sgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb,
x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparms, params, work,
iwork, info )

call dgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb,
x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparms, params, work,
iwork, info )

call cgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb,
x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparms, params, work,
rwork, info )

call zgbrfsx( trans, equed, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, r, c, b, ldb,
x, ldx, rcond, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparms, params, work,
rwork, info )
```

C:

```

lapack_int LAPACKE_sgbrfsx( int matrix_layout, char trans, char equed, lapack_int n,
                            lapack_int kl, lapack_int ku, lapack_int nrhs, const float* ab, lapack_int ldab,
                            const float* afb, lapack_int ldafb, const lapack_int* ipiv, const float* r,
                            const float* c, const float* b, lapack_int ldb, float* x, lapack_int ldx, float*
                            rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp,
                            lapack_int nparams, float* params );

lapack_int LAPACKE_dgbrfsx( int matrix_layout, char trans, char equed, lapack_int n,
                            lapack_int kl, lapack_int ku, lapack_int nrhs, const double* ab, lapack_int ldab,
                            const double* afb, lapack_int ldafb, const lapack_int* ipiv, const double* r,
                            const double* c, const double* b, lapack_int ldb, double* x, lapack_int ldx, double*
                            rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double*
                            err_bnds_comp, lapack_int nparams, double* params );

lapack_int LAPACKE_cgbrfsx( int matrix_layout, char trans, char equed, lapack_int n,
                            lapack_int kl, lapack_int ku, lapack_int nrhs, const lapack_complex_float* ab,
                            lapack_int ldab, const lapack_complex_float* afb, lapack_int ldafb, const lapack_int* ipiv,
                            const float* r, const float* c, const lapack_complex_float* b, lapack_int ldb,
                            lapack_complex_float* x, lapack_int ldx, float* rcond, float* berr, lapack_int
                            n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float*
                            params );

lapack_int LAPACKE_zgbrfsx( int matrix_layout, char trans, char equed, lapack_int n,
                            lapack_int kl, lapack_int ku, lapack_int nrhs, const lapack_complex_double* ab,
                            lapack_int ldab, const lapack_complex_double* afb, lapack_int ldafb, const lapack_int* ipiv,
                            const double* r, const double* c, const lapack_complex_double* b, lapack_int ldb,
                            lapack_complex_double* x, lapack_int ldx, double* rcond, double* berr, lapack_int
                            n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double*
                            params );

```

Include Files

- Fortran: mkl.fi
 - Fortran 95: lapack.f90
 - C: mkl.h

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed*, *r*, and *c* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

trans CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If $trans = 'N'$, the system has the form A^*x

If $\text{trans} = \text{'T'}$, the system has the form $A^{**} T^* X = B$ (Transpose).

If $\text{trans} = \text{'C'}$, the system has the form $A^{**} H^* X = B$ (Conjugate transpose = Transpose).

equed

CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.

Specifies the form of equilibration that was done to A before calling this routine.

If $\text{equed} = \text{'N'}$, no equilibration was done.

If $\text{equed} = \text{'R'}$, row equilibration was done, that is, A has been premultiplied by $\text{diag}(r)$.

If $\text{equed} = \text{'C'}$, column equilibration was done, that is, A has been postmultiplied by $\text{diag}(c)$.

If $\text{equed} = \text{'B'}$, both row and column equilibration was done, that is, A has been replaced by $\text{diag}(r)^* A^* \text{diag}(c)$. The right-hand side B has been changed accordingly.

n

INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.

k1

INTEGER. The number of subdiagonals within the band of A ; $k1 \geq 0$.

ku

INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.

ab, afb, b, work

REAL for sgbrfsx

DOUBLE PRECISION for dgbrfsx

COMPLEX for cgbrfsx

DOUBLE COMPLEX for zgbrfsx.

Arrays: $ab(1dab, *)$, $afb(1dafb, *)$, $b(1db, *)$, $work(*)$.

The array ab contains the original matrix A in band storage, in rows 1 to $k1 + ku + 1$. The j -th column of A is stored in the j -th column of the array ab as follows:

$ab(ku+1+i-j, j) = A(i, j)$ for $\max(1, j-ku) \leq i \leq \min(n, j+k1)$.

The array afb contains details of the LU factorization of the banded matrix A as computed by [?gbtrf](#). U is stored as an upper triangular banded matrix with $k1 + ku$ superdiagonals in rows 1 to $k1 + ku + 1$. The multipliers used during the factorization are stored in rows $k1 + ku + 2$ to $2*k1 + ku + 1$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

1dab

INTEGER. The leading dimension of the array ab ; $1dab \geq k1 + ku + 1$.

1dafb

INTEGER. The leading dimension of the array afb ; $1dafb \geq 2*k1 + ku + 1$.

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, n)$. Contains the pivot indices as computed by [?gbtrf](#); for row $1 \leq i \leq n$, row i of the matrix was interchanged with row $ipiv(i)$.

r, c

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: $r(n)$, $c(n)$. The array r contains the row scale factors for A , and the array c contains the column scale factors for A .

If $equed = 'R'$ or $'B'$, A is multiplied on the left by $diag(r)$; if $equed = 'N'$ or $'C'$, r is not accessed.

If $equed = 'R'$ or $'B'$, each element of r must be positive.

If $equed = 'C'$ or $'B'$, A is multiplied on the right by $diag(c)$; if $equed = 'N'$ or $'R'$, c is not accessed.

If $equed = 'C'$ or $'B'$, each element of c must be positive.

Each element of r or c should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

*ldb*INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.*x*

REAL for sgbrfsx

DOUBLE PRECISION for dgbrfsx

COMPLEX for cgbrfsx

DOUBLE COMPLEX for zgbrfsx.

Array, DIMENSION ($lidx, *$).

The solution matrix X as computed by [sgbtrs/dgbtrs](#) for real flavors or [cgbtrs/zgbtrs](#) for complex flavors.

*lidx*INTEGER. The leading dimension of the output array x ; $lidx \geq \max(1, n)$.*n_err_bnds*

INTEGER. Number of error bounds to return for each right-hand side and each type (normwise or componentwise). See err_bnds_norm and err_bnds_comp descriptions in *Output Arguments* section below.

nparams

INTEGER. Specifies the number of parameters set in $params$. If ≤ 0 , the $params$ array is never referenced and default values are used.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION $nparams$. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to $nparams$ are accessed; defaults are used

for higher-numbered parameters. If defaults are acceptable, you can pass `nparams = 0`, which prevents the source code from accessing the `params` argument.

`params(la_linrx_itref_i = 1)` : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
------	--

=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support DOUBLE PRECISION.
------	--

(Other values are reserved for future use.)

`params(la_linrx_ithresh_i = 2)` : Maximum number of residual computations allowed for refinement.

Default	10
---------	----

Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in <code>err_bnds_norm</code> and <code>err_bnds_comp</code> may no longer be trustworthy.
------------	---

`params(la_linrx_cwise_i = 3)` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

`iwork` INTEGER. Workspace array, DIMENSION at least `max(1, n)`; used in real flavors only.

`rwork` REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Workspace array, DIMENSION at least `max(1, 3*n)`; used in complex flavors only.

Output Parameters

`x` REAL for sgbrfsx
DOUBLE PRECISION for dgbrfsx
COMPLEX for cgbrfsx
DOUBLE COMPLEX for zgbrfsx.
The improved solution matrix X.

`rcond` REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

err_bnds_norm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION $(nrhs, n_err_bnds)$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the i -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in $err_bnds_norm(i, :)$ corresponds to the i -th right-hand side.

The second index in $err_bnds_norm(:, err)$ contains the following three fields:

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors.

err=2

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is

"guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION ($\text{nrhs}, n_{\text{err_bnds}}$). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{\text{true}_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params}(3) = 0.0$), then `err_bnds_comp` is not accessed. If $n_{\text{err_bnds}} < 3$, then at most the first $(:, n_{\text{err_bnds}})$ entries are returned.

The first index in `err_bnds_comp(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:, err)` contains the following three fields:

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\text{sqrt}(n) * \text{slamch}(\epsilon)$ for single precision flavors and $\text{sqrt}(n) * \text{dlamch}(\epsilon)$ for double precision flavors.

`err=2`

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\text{sqrt}(n) * \text{slamch}(\epsilon)$ for single precision flavors and $\text{sqrt}(n) * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\text{sqrt}(n) * \text{slamch}(\epsilon)$ for single precision flavors and $\text{sqrt}(n) * \text{dlamch}(\epsilon)$ for double precision

flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s^*(\alpha * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $\alpha * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

<i>params</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Output parameter only if the input contains erroneous values, namely, in <i>params</i> (1), <i>params</i> (2), <i>params</i> (3). In such a case, the corresponding elements of <i>params</i> are filled with default values on output.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. The solution to every right-hand side is guaranteed. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned. If <i>info</i> = $n+j$: The solution corresponding to the <i>j</i> -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides <i>k</i> with <i>k</i> > <i>j</i> may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested <i>params</i> (3) = 0.0, then the <i>j</i> -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest <i>j</i> such that <i>err_bnds_norm(j, 1)</i> = 0.0 or <i>err_bnds_comp(j, 1)</i> = 0.0. See the definition of <i>err_bnds_norm(, 1)</i> and <i>err_bnds_comp(, 1)</i> . To get information about all of the right-hand sides, check <i>err_bnds_norm</i> or <i>err_bnds_comp</i> .

?gtrfs

Refines the solution of a system of linear equations with a tridiagonal matrix and estimates its error.

Syntax

FORTRAN 77:

```
call sgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
ferr, berr, work, iwork, info )

call dgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
ferr, berr, work, iwork, info )

call cgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
ferr, berr, work, rwork, info )

call zgtrfs( trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x, ldx,
ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gtrfs( dl, d, du, dlf, df, duf, du2, ipiv, b, x [,trans] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_sgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const float* dl, const float* d, const float* du, const float* dlf, const float* df, const float* duf, const float* du2, const lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );  
lapack_int LAPACKE_dgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const double* dl, const double* d, const double* du, const double* dlf, const double* df, const double* duf, const double* du2, const lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );  
lapack_int LAPACKE_cgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const lapack_complex_float* dl, const lapack_complex_float* d, const lapack_complex_float* du, const lapack_complex_float* dlf, const lapack_complex_float* df, const lapack_complex_float* duf, const lapack_complex_float* du2, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );  
lapack_int LAPACKE_zgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const lapack_complex_double* dl, const lapack_complex_double* d, const lapack_complex_double* du, const lapack_complex_double* dlf, const lapack_complex_double* df, const lapack_complex_double* duf, const lapack_complex_double* du2, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^T X = B$ or $A^H X = B$ with a tridiagonal matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| / |a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| / |b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?gttrf](#)
- call the solver routine [?gttrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

trans

CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If $\text{trans} = \text{'N'}$, the system has the form $A*X = B$.

If $\text{trans} = \text{'T'}$, the system has the form $A^T*X = B$.

If $\text{trans} = \text{'C'}$, the system has the form $A^H*X = B$.

n

INTEGER. The order of the matrix A ; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides, that is, the number of columns of the matrix B ; $nrhs \geq 0$.

dl,d,du,dlf,

df,duf,du2,

b,x,work

REAL for sgtrfs

DOUBLE PRECISION for dgtrfs

COMPLEX for cgtrfs

DOUBLE COMPLEX for zgtrfs.

Arrays:

dl, dimension $(n - 1)$, contains the subdiagonal elements of A .

d, dimension (n) , contains the diagonal elements of A .

du, dimension $(n - 1)$, contains the superdiagonal elements of A .

dlf, dimension $(n - 1)$, contains the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A as computed by [?gttrf](#).

df, dimension (n) , contains the n diagonal elements of the upper triangular matrix U from the LU factorization of A .

duf, dimension $(n - 1)$, contains the $(n - 1)$ elements of the first superdiagonal of U .

du2, dimension $(n - 2)$, contains the $(n - 2)$ elements of the second superdiagonal of U .

b(1:db, nrhs) contains the right-hand side matrix B .

x(1:dx, nrhs) contains the solution matrix X , as computed by [?gttrs](#).

work()* is a workspace array; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldb

INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ldx

INTEGER. The leading dimension of *x*; $ldx \geq \max(1, n)$.

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The *ipiv* array, as returned by [?gttrf](#).

iwork

INTEGER. Workspace array, DIMENSION (n) . Used for real flavors only.

rwork

REAL for cgtrfs

DOUBLE PRECISION for zgtrfs.

Workspace array, DIMENSION (n) . Used for complex flavors only.

Output Parameters

<i>x</i>	The refined solution matrix X .
<i>ferr</i> , <i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.
	Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gtrfs` interface are as follows:

<i>d1</i>	Holds the vector of length $(n-1)$.
<i>d</i>	Holds the vector of length n .
<i>du</i>	Holds the vector of length $(n-1)$.
<i>d1f</i>	Holds the vector of length $(n-1)$.
<i>df</i>	Holds the vector of length n .
<i>duf</i>	Holds the vector of length $(n-1)$.
<i>du2</i>	Holds the vector of length $(n-2)$.
<i>ipiv</i>	Holds the vector of length n .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.

?porfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

FORTRAN 77:

```

call sporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, iwork,
info )

call dporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, iwork,
info )

call cporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, rwork,
info )

call zporfs( uplo, n, nrhs, a, lda, af, ldaf, b, ldb, x, ldx, ferr, berr, work, rwork,
info )

```

Fortran 95:

```
call porfs( a, af, b, x [,uplo] [,ferr] [,berr] [,info] )
```

C:

```

lapack_int LAPACKE_sporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* a, lapack_int lda, const float* af, lapack_int ldaf, const float* b,
lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_dporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* a, lapack_int lda, const double* af, lapack_int ldaf, const double* b,
lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );

lapack_int LAPACKE_cporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af,
lapack_int ldaf, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float*
x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_zporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af,
lapack_int ldaf, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double*
x, lapack_int ldx, double* ferr, double* berr );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a symmetric (Hermitian) positive definite matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?potrf](#)
- call the solver routine [?potrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; <i>n</i> \geq 0.
<i>nrhs</i>	INTEGER. The number of right-hand sides; <i>nrhs</i> \geq 0.
<i>a,af,b,x,work</i>	REAL for <i>sporfs</i> DOUBLE PRECISION for <i>dporfs</i> COMPLEX for <i>cporfs</i> DOUBLE COMPLEX for <i>zporfs</i> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the original matrix <i>A</i> , as supplied to ?potrf . <i>af</i> (<i>ldaf</i> ,*) contains the factored matrix <i>A</i> , as returned by ?potrf . <i>b</i> (<i>ldb</i> ,*) contains the right-hand side matrix <i>B</i> . <i>x</i> (<i>idx</i> ,*) contains the solution matrix <i>X</i> . <i>work</i> (*) is a workspace array. The second dimension of <i>a</i> and <i>af</i> must be at least $\max(1, n)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of <i>af</i> ; <i>ldaf</i> $\geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$.
<i>idx</i>	INTEGER. The leading dimension of <i>x</i> ; <i>idx</i> $\geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>cporfs</i> DOUBLE PRECISION for <i>zporfs</i> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr, berr</i>	REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `porfs` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>af</i>	Holds the matrix AF of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?porfsx

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric/Hermitian positive-definite matrix A and provides error bounds and backward error estimates.

Syntax

FORTRAN 77:

```
call sporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, idx, rcond, berr,
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
call dporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, idx, rcond, berr,
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
```

```
call cporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, idx, rcond, berr,  
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )  
call zporfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, s, b, ldb, x, idx, rcond, berr,  
n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

C:

```
lapack_int LAPACKE_sporfsx( int matrix_layout, char uplo, char equed, lapack_int n,  
lapack_int nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf,  
const float* s, const float* b, lapack_int ldb, float* x, lapack_int idx, float*  
rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp,  
lapack_int nparams, float* params );  
  
lapack_int LAPACKE_dporfsx( int matrix_layout, char uplo, char equed, lapack_int n,  
lapack_int nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf,  
const double* s, const double* b, lapack_int ldb, double* x, lapack_int idx, double*  
rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double*  
err_bnds_comp, lapack_int nparams, double* params );  
  
lapack_int LAPACKE_cporfsx( int matrix_layout, char uplo, char equed, lapack_int n,  
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda,  
const lapack_complex_float* af, lapack_int ldaf, const float* s,  
const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int idx,  
float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float*  
err_bnds_comp, lapack_int nparams, float* params );  
  
lapack_int LAPACKE_zporfsx( int matrix_layout, char uplo, char equed, lapack_int n,  
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda,  
const lapack_complex_double* af, lapack_int ldaf, const double* s,  
const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int idx,  
double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm,  
double* err_bnds_comp, lapack_int nparams, double* params );
```

Include Files

- **Fortran:** mkl.fi
- **Fortran 95:** lapack.f90
- **C:** mkl.h

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters *equed* and *s* below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If $uplo = 'U'$, the upper triangle of A is stored.

If $uplo = 'L'$, the lower triangle of A is stored.

$equed$

CHARACTER*1. Must be ' N ' or ' Y '.

Specifies the form of equilibration that was done to A before calling this routine.

If $equed = 'N'$, no equilibration was done.

If $equed = 'Y'$, both row and column equilibration was done, that is, A has been replaced by $diag(s) * A * diag(s)$. The right-hand side B has been changed accordingly.

n

INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.

$nrhs$

INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.

$a, af, b, work$

REAL for `sporfsx`

DOUBLE PRECISION for `dporfsx`

COMPLEX for `cporfsx`

DOUBLE COMPLEX for `zporfsx`.

Arrays: $a(1da, *), af(1daf, *), b(1db, *), work(*)$.

The array a contains the symmetric/Hermitian matrix A as specified by $uplo$. If $uplo = 'U'$, the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A and the strictly lower triangular part of a is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A and the strictly upper triangular part of a is not referenced. The second dimension of a must be at least $\max(1, n)$.

The array af contains the triangular factor L or U from the Cholesky factorization $A = U^{**}T^{*}U$ or $A = L^{*}L^{**}T$ as computed by `spotrf` for real flavors or `dpotrf` for complex flavors.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

$1da$

INTEGER. The leading dimension of a ; $1da \geq \max(1, n)$.

$1daf$

INTEGER. The leading dimension of af ; $1daf \geq \max(1, n)$.

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (n). The array s contains the scale factors for A .

If $equed = 'N'$, s is not accessed.

If $equed = 'Y'$, each element of s must be positive.

Each element of s should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

`ldb` INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

`x` REAL for `sporfsx`

DOUBLE PRECISION for `dporfsx`

COMPLEX for `cporfsx`

DOUBLE COMPLEX for `zporfsx`.

Array, DIMENSION ($lidx, *$).

The solution matrix X as computed by `?potrs`

`lidx` INTEGER. The leading dimension of the output array x ; $lidx \geq \max(1, n)$.

`n_err_bnds` INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See `err_bnds_norm` and `err_bnds_comp` descriptions in *Output Arguments* section below.

`nparams` INTEGER. Specifies the number of parameters set in `params`. If ≤ 0 , the `params` array is never referenced and default values are used.

`params` REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION $nparams$. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to $nparams$ are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass $nparams = 0$, which prevents the source code from accessing the `params` argument.

`params(la_linx_itref_i = 1)` : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

<code>=0.0</code>	No refinement is performed and no error bounds are computed.
-------------------	--

<code>=1.0</code>	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support DOUBLE PRECISION.
-------------------	--

(Other values are reserved for future use.)

`params(la_linx_ithresh_i = 2)` : Maximum number of residual computations allowed for refinement.

Default	10
---------	----

	Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than <i>LU</i> . If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.
		<i>params(la_linrx_cwise_i = 3)</i> : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).
<i>iwork</i>		INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.
<i>rwork</i>		REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Workspace array, DIMENSION at least $\max(1, 3*n)$; used in complex flavors only.

Output Parameters

<i>x</i>	REAL for <i>sporfsx</i> DOUBLE PRECISION for <i>dporfsx</i> COMPLEX for <i>cporfsx</i> DOUBLE COMPLEX for <i>zporfsx</i> . The improved solution matrix <i>X</i> .
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision, in particular, if <i>rcond</i> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.
<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector <i>x(j)</i> , that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes <i>x(j)</i> an exact solution.
<i>err_bnds_norm</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (<i>nrhs, n_err_bnds</i>). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows: Normwise relative error in the <i>i</i> -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i, :)` corresponds to the *i*-th right-hand side.

The second index in `err_bnds_norm(:, err)` contains the following three fields:

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

`err=2`

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix Z.

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (`nrhs, n_err_bnds`). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params}(3) = 0.0$), then err_bnds_comp is not accessed. If $n_{\text{err_bnds}} < 3$, then at most the first $(:, n_{\text{err_bnds}})$ entries are returned.

The first index in $\text{err_bnds_comp}(i, :)$ corresponds to the i -th right-hand side.

The second index in $\text{err_bnds_comp}(:, \text{err})$ contains the following three fields:

$\text{err}=1$	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
$\text{err}=2$	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
$\text{err}=3$	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z . Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Output parameter only if the input contains erroneous values, namely, in $\text{params}(1)$, $\text{params}(2)$, $\text{params}(3)$. In such a case, the corresponding elements of params are filled with default values on output.

info

INTEGER. If $\text{info} = 0$, the execution is successful. The solution to every right-hand side is guaranteed.

If $\text{info} = -i$, the i -th parameter had an illegal value.

If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $\text{rcond} = 0$ is returned.

If $\text{info} = n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $\text{params}(3) = 0.0$, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that $\text{err_bnds_norm}(j, 1) = 0.0$ or $\text{err_bnds_comp}(j, 1) = 0.0$). See the definition of $\text{err_bnds_norm}(:, 1)$ and $\text{err_bnds_comp}(:, 1)$. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp .

?pprfs

Refines the solution of a system of linear equations with a packed symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

FORTRAN 77:

```
call spprfs( uplo, n, nrhs, ap, afp, b, ldb, x, idx, ferr, berr, work, iwork, info )
call dpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, idx, ferr, berr, work, iwork, info )
call cpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, idx, ferr, berr, work, rwork, info )
call zpprfs( uplo, n, nrhs, ap, afp, b, ldb, x, idx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call pprfs( ap, afp, b, x [,uplo] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_spprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* ap, const float* afp, const float* b, lapack_int ldb, float* x,
lapack_int idx, float* ferr, float* berr );
lapack_int LAPACKE_dpprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* ap, const double* afp, const double* b, lapack_int ldb, double* x,
lapack_int idx, double* ferr, double* berr );
lapack_int LAPACKE_cpprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* ap, const lapack_complex_float* afp,
const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int idx,
float* ferr, float* berr );
lapack_int LAPACKE_zpprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* ap, const lapack_complex_double* afp,
const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int idx,
double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90

- C: mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a packed symmetric (Hermitian)positive definite matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution

$$\|x - x_e\|_\infty / \|x\|_\infty$$

where x_e is the exact solution.

Before calling this routine:

- call the factorization routine [?pptrf](#)
- call the solver routine [?ptrfs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <i>uplo</i> = 'U', the upper triangle of A is stored.
	If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap, afp, b, x, work</i>	REAL for spprfs DOUBLE PRECISION for dpprfs COMPLEX for cprfs DOUBLE COMPLEX for zpprfs. Arrays: <i>ap</i> (*) contains the original packed matrix A , as supplied to ?pptrf . <i>afp</i> (*) contains the factored packed matrix A , as returned by ?pptrf . <i>b</i> (<i>ldb</i> , *) contains the right-hand side matrix B . <i>x</i> (<i>ldx</i> , *) contains the solution matrix X . <i>work</i> (*) is a workspace array. The dimension of arrays <i>ap</i> and <i>afp</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>cpprfs</i> DOUBLE PRECISION for <i>zpprfs</i> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr</i> , <i>berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.
	Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *pprfs* interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>afp</i>	Holds the array <i>AF</i> of size $(n*(n+1)/2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?pbrfs

Refines the solution of a system of linear equations with a band symmetric (Hermitian) positive-definite matrix and estimates its error.

Syntax

FORTRAN 77:

```
call spbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr,
work, iwork, info )
call dpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr,
work, iwork, info )
call cpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
call zpbrfs( uplo, n, kd, nrhs, ab, ldab, afb, ldafb, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
```

Fortran 95:

```
call pbrfs( ab, afb, b, x [,uplo] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_spbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
lapack_int nrhs, const float* ab, lapack_int ldab, const float* afb, lapack_int ldafb,
const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );
lapack_int LAPACKE_dpbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
lapack_int nrhs, const double* ab, lapack_int ldab, const double* afb, lapack_int ldafb,
const double* b, lapack_int ldb, double* x, lapack_int ldx, double* ferr,
double* berr );
lapack_int LAPACKE_cpbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
lapack_int nrhs, const lapack_complex_float* ab, lapack_int ldab,
const lapack_complex_float* afb, lapack_int ldafb, const lapack_complex_float* b,
lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
lapack_int LAPACKE_zpbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
lapack_int nrhs, const lapack_complex_double* ab, lapack_int ldab,
const lapack_complex_double* afb, lapack_int ldafb, const lapack_complex_double* b,
lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a symmetric (Hermitian) positive definite band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?pbtrf](#)
- call the solver routine [?pbtrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab,afb,b,x,work</i>	REAL for spbrfs DOUBLE PRECISION for dpbrfs COMPLEX for cpbrfs DOUBLE COMPLEX for zpbrfs. Arrays: <i>ab</i> (<i>ldab</i> ,*) contains the original band matrix A , as supplied to ?pbtrf . <i>afb</i> (<i>ldafb</i> ,*) contains the factored band matrix A , as returned by ?pbtrf . <i>b</i> (<i>ldb</i> ,*) contains the right-hand side matrix B . <i>x</i> (<i>ldx</i> ,*) contains the solution matrix X . <i>work</i> (*) is a workspace array. The second dimension of <i>ab</i> and <i>afb</i> must be at least $\max(1, n)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; <i>ldab</i> $\geq kd + 1$.

<i>ldafb</i>	INTEGER. The leading dimension of <i>afb</i> ; $ldafb \geq kd + 1$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for cpbrfs DOUBLE PRECISION for zpbrfs. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr</i> , <i>berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors.
	Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbrfs` interface are as follows:

<i>ab</i>	Holds the array <i>A</i> of size $(kd+1, n)$.
<i>afb</i>	Holds the array <i>AF</i> of size $(kd+1, n)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $8n*kd$ floating-point operations (for real flavors) or $32n*kd$ operations (for complex flavors). In addition, each step of iterative refinement involves $12n*kd$ operations (for real flavors) or $48n*kd$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4n*kd$ floating-point operations for real flavors or $16n*kd$ for complex flavors.

?ptrfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal matrix and estimates its error.

Syntax

FORTRAN 77:

```
call sptrfs( n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, info )
call dptrfs( n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, info )
call cptrfs( uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
call zptrfs( uplo, n, nrhs, d, e, df, ef, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
```

Fortran 95:

```
call ptrfs( d, df, e, ef, b, x [,ferr] [,berr] [,info] )
call ptrfs( d, df, e, ef, b, x [,uplo] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_sptrfs( int matrix_layout, lapack_int n, lapack_int nrhs,
const float* d, const float* e, const float* df, const float* ef, const float* b,
lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );
lapack_int LAPACKE_dptrfs( int matrix_layout, lapack_int n, lapack_int nrhs,
const double* d, const double* e, const double* df, const double* ef, const double* b,
lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );
lapack_int LAPACKE_cptrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* d, const lapack_complex_float* e, const float* df,
const lapack_complex_float* ef, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
lapack_int LAPACKE_zptrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* d, const lapack_complex_double* e, const double* df,
const lapack_complex_double* ef, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ with a symmetric (Hermitian) positive definite tridiagonal matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$|\delta a_{ij}| \leq \beta |a_{ij}|$, $|\delta b_i| \leq \beta |b_i|$ such that $(A + \delta A)x = (b + \delta b)$.

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?pttrf](#)
 - call the solver routine [?pttrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo

CHARACTER*1. Used for complex flavors only. Must be 'U' or 'L'.

Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and how A is factored:

If $uplo = 'U'$, the array e stores the superdiagonal of A , and A is factored as $U^H * D * U$.

If $\text{uplo} = \text{'L'}$, the array e stores the subdiagonal of A , and A is factored as $L \cdot D \cdot L^H$.

n

INTEGER. The order of the matrix A ; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; $nrhs \geq 0$.

d, df, rwork

REAL for single precision flavors **DOUBLE PRECISION** for double precision flavors

Arrays: $d(n)$, $df(n)$, $rwork(n)$.

The array d contains the n diagonal elements of the tridiagonal matrix A .

The array df contains the n diagonal elements of the diagonal matrix D from the factorization of A as computed by [?pttrf](#).

The array *rwork* is a workspace array used for complex flavors only.

e,ef,b,x,work

REAL for sptrfs

DOUBLE PRECISION for dptrfs

COMPLEX for cptrf*s*

DOUBLE COMPLEX for zptrfs.

Arrays: $e(n-1)$, $ef(n-1)$, $b(ldb, nrhs)$, $x(lidx, nrhs)$, $work(*)$.

The array e contains the $(n - 1)$ off-diagonal elements of the tridiagonal matrix A (see *uplo*).

The array *ef* contains the $(n - 1)$ off-diagonal elements of the unit bidiagonal factor *U* or *L* from the factorization computed by [?pttrf](#) (see *uplo*).

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

The array *x* contains the solution matrix *X* as computed by [?pttrs](#).

The array *work* is a workspace array. The dimension of *work* must be at least $2 * n$ for real flavors, and at least *n* for complex flavors.

ldb INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ldx INTEGER. The leading dimension of *x*; $ldx \geq \max(1, n)$.

Output Parameters

x The refined solution matrix *X*.

ferr, *berr* REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ptrfs` interface are as follows:

d Holds the vector of length *n*.

df Holds the vector of length *n*.

e Holds the vector of length $(n-1)$.

ef Holds the vector of length $(n-1)$.

b Holds the matrix *B* of size $(n, nrhs)$.

x Holds the matrix *X* of size $(n, nrhs)$.

ferr Holds the vector of length $(nrhs)$.

berr Holds the vector of length $(nrhs)$.

uplo Used in complex flavors only. Must be '`U`' or '`L`'. The default value is '`U`'.

?syrfs

Refines the solution of a system of linear equations with a symmetric matrix and estimates its error.

Syntax**FORTRAN 77:**

```
call ssyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call dsyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork, info )
call csyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zsyrfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call syrfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_ssyrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* a, lapack_int lda, const float* af, lapack_int ldaf, const lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );
lapack_int LAPACKE_dsyrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* a, lapack_int lda, const double* af, lapack_int ldaf, const lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );
lapack_int LAPACKE_csyrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af,
lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
lapack_int LAPACKE_zsyrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af,
lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a symmetric full-storage matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?sytrf](#)
- call the solver routine [?sytrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored. If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; <i>n</i> ≥ 0.
<i>nrhs</i>	INTEGER. The number of right-hand sides; <i>nrhs</i> ≥ 0.
<i>a,af,b,x,work</i>	REAL for ssyrf DOUBLE PRECISION for dsyrf COMPLEX for csyrf DOUBLE COMPLEX for zsyrf. Arrays: <i>a</i> (<i>lda</i> ,*) contains the original matrix <i>A</i> , as supplied to ?sytrf . <i>af</i> (<i>ldaf</i> ,*) contains the factored matrix <i>A</i> , as returned by ?sytrf . <i>b</i> (<i>ldb</i> ,*) contains the right-hand side matrix <i>B</i> . <i>x</i> (<i>idx</i> ,*) contains the solution matrix <i>X</i> . <i>work</i> (*) is a workspace array. The second dimension of <i>a</i> and <i>af</i> must be at least $\max(1, n)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; <i>lda</i> ≥ $\max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of <i>af</i> ; <i>ldaf</i> ≥ $\max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; <i>ldb</i> ≥ $\max(1, n)$.
<i>idx</i>	INTEGER. The leading dimension of <i>x</i> ; <i>idx</i> ≥ $\max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sytrf .
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for csyrf DOUBLE PRECISION for zsyrf.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix X .
<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.
	Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `syrfs` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>af</i>	Holds the matrix AF of size (n, n) .
<i>ipiv</i>	Holds the vector of length n .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?syrfsx

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric indefinite matrix A and provides error bounds and backward error estimates.

Syntax**FORTRAN 77:**

```
call ssyrfxs( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
call dsyrfxs( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork, info )
call csyrfxs( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
call zsyrfxs( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

C:

```
lapack_int LAPACKE_ssyrfsx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf,
const lapack_int* ipiv, const float* s, const float* b, lapack_int ldb, float* x,
lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float*
err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACKE_dsyrfxs( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf,
const lapack_int* ipiv, const double* s, const double* b, lapack_int ldb, double* x,
lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds, double*
err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );

lapack_int LAPACKE_csyrfxs( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda,
const lapack_complex_float* af, lapack_int ldaf, const lapack_int* ipiv, const float* s,
const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float*
err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACKE_zsyrfxs( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda,
const lapack_complex_double* af, lapack_int ldaf, const lapack_int* ipiv, const double* s,
const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm,
double* err_bnds_comp, lapack_int nparams, double* params );
```

Include Files

- **Fortran:** mkl.fi
- **Fortran 95:** lapack.f90
- **C:** mkl.h

Description

The routine improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters `equed` and `s` below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '.
	Indicates whether the upper or lower triangular part of A is stored: If <code>uplo</code> = ' <code>U</code> ', the upper triangle of A is stored. If <code>uplo</code> = ' <code>L</code> ', the lower triangle of A is stored.
<code>equed</code>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>Y</code> '.
	Specifies the form of equilibration that was done to A before calling this routine. If <code>equed</code> = ' <code>N</code> ', no equilibration was done. If <code>equed</code> = ' <code>Y</code> ', both row and column equilibration was done, that is, A has been replaced by $\text{diag}(s) * A * \text{diag}(s)$. The right-hand side B has been changed accordingly.
<code>n</code>	INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.
<code>a, af, b, work</code>	REAL for <code>ssyrfxsx</code> DOUBLE PRECISION for <code>dsyrfxsx</code> COMPLEX for <code>csyrfxsx</code> DOUBLE COMPLEX for <code>zsyrfxsx</code> . Arrays: <code>a(lda,*)</code> , <code>af(ldaf,*)</code> , <code>b(ldb,*)</code> , <code>work(*)</code> . The array <code>a</code> contains the symmetric/Hermitian matrix A as specified by <code>uplo</code> . If <code>uplo</code> = ' <code>U</code> ', the leading n -by- n upper triangular part of <code>a</code> contains the upper triangular part of the matrix A and the strictly lower triangular part of <code>a</code> is not referenced. If <code>uplo</code> = ' <code>L</code> ', the leading n -by- n lower triangular part of <code>a</code> contains the lower triangular part of the matrix A and the strictly upper triangular part of <code>a</code> is not referenced. The second dimension of <code>a</code> must be at least $\max(1, n)$. The array <code>af</code> contains the triangular factor L or U from the Cholesky factorization $A = U^* T^* U$ or $A = L^* L^* T$ as computed by <code>ssytrf</code> for real flavors or <code>dsytrf</code> for complex flavors.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

work()* is a workspace array. The dimension of *work* must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of af ; $ldaf \geq \max(1, n)$.

ipiv INTEGER.

Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of D as determined by [ssytrf](#) for real flavors or [dsytrf](#) for complex flavors.

s REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (n). The array s contains the scale factors for A .

If *equed* = 'N', s is not accessed.

If *equed* = 'Y', each element of s must be positive.

Each element of s should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

x REAL for [ssyrfxsx](#)

DOUBLE PRECISION for [dsyrfxsx](#)

COMPLEX for [csyrfxsx](#)

DOUBLE COMPLEX for [zsyrfxsx](#).

Array, DIMENSION ($l dx, *$).

The solution matrix X as computed by [?sytrs](#)

l dx INTEGER. The leading dimension of the output array x ; $l dx \geq \max(1, n)$.

n_err_bnds INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

nparams INTEGER. Specifies the number of parameters set in *params*. If ≤ 0 , the *params* array is never referenced and default values are used.

params REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

`Array, DIMENSION nparams.` Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

`params(la_linrx_itref_i = 1)` : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
------	--

=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support DOUBLE PRECISION.
------	--

(Other values are reserved for future use.)

`params(la_linrx_ithresh_i = 2)` : Maximum number of residual computations allowed for refinement.

Default	10
---------	----

Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in <code>err_bnds_norm</code> and <code>err_bnds_comp</code> may no longer be trustworthy.
------------	---

`params(la_linrx_cwise_i = 3)` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

`iwork` INTEGER. Workspace array, DIMENSION at least `max(1, n)`; used in real flavors only.

`rwork` REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION at least `max(1, 3*n)`; used in complex flavors only.

Output Parameters

`x` REAL for `ssyrfxsx`

DOUBLE PRECISION for `dsyrfxsx`

COMPLEX for `csyrfxsx`

DOUBLE COMPLEX for `zsyrfxsx`.

The improved solution matrix X .

<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.						
	Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.						
<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.						
<i>err_bnds_norm</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION $(nrhs, n_{err_bnds})$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows: Normwise relative error in the i -th solution vector $\frac{\max_j X_{true_{ji}} - X_{ji} }{\max_j X_{ji} }$ The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned. The first index in $err_bnds_norm(i, :)$ corresponds to the i -th right-hand side. The second index in $err_bnds_norm(:, err)$ contains the following three fields: <table border="0"><tr><td style="vertical-align: top;"><i>err=1</i></td><td>"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors.</td></tr><tr><td style="vertical-align: top;"><i>err=2</i></td><td>"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.</td></tr><tr><td style="vertical-align: top;"><i>err=3</i></td><td>Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold</td></tr></table>	<i>err=1</i>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors.	<i>err=2</i>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.	<i>err=3</i>	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold
<i>err=1</i>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors.						
<i>err=2</i>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.						
<i>err=3</i>	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold						

$\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (`nrhs, n_err_bnds`). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params(3) = 0.0`), then `err_bnds_comp` is not accessed. If $n_err_bnds < 3$, then at most the first $(:, n_err_bnds)$ entries are returned.

The first index in `err_bnds_comp(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:, err)` contains the following three fields:

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

`err=2`

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s^* (a^* \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a^* \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

`params`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Output parameter only if the input contains erroneous values, namely, in `params(1)`, `params(2)`, `params(3)`. In such a case, the corresponding elements of `params` are filled with default values on output.

`info`

INTEGER. If `info` = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If `info` = $-i$, the i -th parameter had an illegal value.

If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; `rcond` = 0 is returned.

If `info` = $n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested `params(3)` = 0.0, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that `err_bnds_norm(j, 1)` = 0.0 or `err_bnds_comp(j, 1)` = 0.0). See the definition of `err_bnds_norm(:, 1)` and `err_bnds_comp(:, 1)`. To get information about all of the right-hand sides, check `err_bnds_norm` or `err_bnds_comp`.

?herfs

Refines the solution of a system of linear equations with a complex Hermitian matrix and estimates its error.

Syntax

FORTRAN 77:

```
call cherfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
call zherfs( uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call herfs( a, af, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_cherfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
                           const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af,
                           lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
                           lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_zherfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
                           const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af,
                           lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
                           lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ with a complex Hermitian full-storage matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?hetrf](#)
- call the solver routine [?hetrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a,af,b,x,work</i>	COMPLEX for cherfs DOUBLE COMPLEX for zherfs. Arrays: <i>a</i> (<i>lda</i> ,*) contains the original matrix A , as supplied to ?hetrf .

af(*ldaf*, *) contains the factored matrix *A*, as returned by [?hetrf](#).

b(*ldb*, *) contains the right-hand side matrix *B*.

x(*ldx*, *) contains the solution matrix *X*.

work(*) is a workspace array.

The second dimension of *a* and *af* must be at least $\max(1, n)$; the second dimension of *b* and *x* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 2*n)$.

lda

INTEGER. The leading dimension of *a*; *lda* $\geq \max(1, n)$.

ldaf

INTEGER. The leading dimension of *af*; *ldaf* $\geq \max(1, n)$.

ldb

INTEGER. The leading dimension of *b*; *ldb* $\geq \max(1, n)$.

ldx

INTEGER. The leading dimension of *x*; *ldx* $\geq \max(1, n)$.

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The *ipiv* array, as returned by [?hetrf](#).

rwork

REAL for *cherfs*

DOUBLE PRECISION for *zherfs*.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

x

The refined solution matrix *X*.

ferr, *berr*

REAL for *cherfs*

DOUBLE PRECISION for *zherfs*.

Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *herfs* interface are as follows:

a

Holds the matrix *A* of size (*n*, *n*).

af

Holds the matrix *AF* of size (*n*, *n*).

ipiv

Holds the vector of length *n*.

<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be ' <i>U</i> ' or ' <i>L</i> '. The default value is ' <i>U</i> '.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is [?ssyrf](#)/[?dsyrf](#)

?herfsx

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric indefinite matrix A and provides error bounds and backward error estimates.

Syntax

FORTRAN 77:

```
call cherfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
call zherfsx( uplo, equed, n, nrhs, a, lda, af, ldaf, ipiv, s, b, ldb, x, ldx, rcond,
berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork, info )
```

C:

```
lapack_int LAPACKE_cherfsx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda,
const lapack_complex_float* af, lapack_int ldaf, const lapack_int* ipiv, const float* s,
const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float*
err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACKE_zherfsx( int matrix_layout, char uplo, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda,
const lapack_complex_double* af, lapack_int ldaf, const lapack_int* ipiv, const double* s,
const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm,
double* err_bnds_comp, lapack_int nparams, double* params );
```

Include Files

- Fortran: mkl.fi

- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite, and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters `equed` and `s` below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '. Indicates whether the upper or lower triangular part of A is stored: If <code>uplo</code> = ' <code>U</code> ', the upper triangle of A is stored. If <code>uplo</code> = ' <code>L</code> ', the lower triangle of A is stored.
<code>equed</code>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>Y</code> '. Specifies the form of equilibration that was done to A before calling this routine. If <code>equed</code> = ' <code>N</code> ', no equilibration was done. If <code>equed</code> = ' <code>Y</code> ', both row and column equilibration was done, that is, A has been replaced by $\text{diag}(s) * A * \text{diag}(s)$. The right-hand side B has been changed accordingly.
<code>n</code>	INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.
<code>a, af, b, work</code>	COMPLEX for <code>cherfsx</code> DOUBLE COMPLEX for <code>zherfsx</code> . Arrays: <code>a(1da,*)</code> , <code>af(1daf,*)</code> , <code>b(1db,*)</code> , <code>work(*)</code> . The array <code>a</code> contains the Hermitian matrix A as specified by <code>uplo</code> . If <code>uplo</code> = ' <code>U</code> ', the leading n -by- n upper triangular part of <code>a</code> contains the upper triangular part of the matrix A and the strictly lower triangular part of <code>a</code> is not referenced. If <code>uplo</code> = ' <code>L</code> ', the leading n -by- n lower triangular part of <code>a</code> contains the lower triangular part of the matrix A and the strictly upper triangular part of <code>a</code> is not referenced. The second dimension of <code>a</code> must be at least $\max(1, n)$. The factored form of the matrix A . The array <code>af</code> contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U^*D^*U^{**}T$ or $A = L^*D^*L^{**}T$ as computed by <code>ssytrf</code> for <code>cherfsx</code> or <code>dsytrf</code> for <code>zherfsx</code> .

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 2*n)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of af ; $ldaf \geq \max(1, n)$.

ipiv INTEGER.

Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of D as determined by [ssytrf](#) for real flavors or [dsytrf](#) for complex flavors.

s REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (n). The array s contains the scale factors for A .

If $equed = 'N'$, s is not accessed.

If $equed = 'Y'$, each element of s must be positive.

Each element of s should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

x COMPLEX for [cherfsx](#)

DOUBLE COMPLEX for [zherfsx](#).

Array, DIMENSION ($lidx, *$).

The solution matrix X as computed by [?hetrs](#)

lidx INTEGER. The leading dimension of the output array x ; $lidx \geq \max(1, n)$.

n_err_bnds INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

nparams INTEGER. Specifies the number of parameters set in *params*. If ≤ 0 , the *params* array is never referenced and default values are used.

params REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION $nparams$. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to $nparams$ are accessed; defaults are used

for higher-numbered parameters. If defaults are acceptable, you can pass $nparams = 0$, which prevents the source code from accessing the *params* argument.

params(la_linx_itref_i = 1) : Whether to perform iterative refinement or not. Default: 1.0 (for *cherfsx*), 1.0D+0 (for *zherfsx*).

=0.0	No refinement is performed and no error bounds are computed.
------	--

=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support DOUBLE PRECISION.
------	--

(Other values are reserved for future use.)

params(la_linx_ithresh_i = 2) : Maximum number of residual computations allowed for refinement.

Default	10
---------	----

Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.
------------	---

params(la_linx_cwise_i = 3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

rwork

REAL for *cherfsx*

DOUBLE PRECISION for *zherfsx*.

Workspace array, DIMENSION at least $\max(1, 3*n)$.

Output Parameters

x

COMPLEX for *cherfsx*

DOUBLE COMPLEX for *zherfsx*.

The improved solution matrix *X*.

rcond

REAL for *cherfsx*

DOUBLE PRECISION for *zherfsx*.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix *A* after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr

REAL for *cherfsx*

DOUBLE PRECISION for zherfsx.

Array, DIMENSION at least $\max(1, \text{nrhs})$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

err_bnds_norm

REAL for cherfsx

DOUBLE PRECISION for zherfsx.

Array, DIMENSION $(\text{nrhs}, n_{\text{err_bnds}})$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the i -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm*($i, :$) corresponds to the i -th right-hand side.

The second index in *err_bnds_norm*($:, err$) contains the following three fields:

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for cherfsx and $\sqrt{n} * \text{dlamch}(\epsilon)$ for zherfsx.

err=2

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for cherfsx and $\sqrt{n} * \text{dlamch}(\epsilon)$ for zherfsx. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for cherfsx and $\sqrt{n} * \text{dlamch}(\epsilon)$ for zherfsx to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

err_bnds_comp

REAL for cherfsx

DOUBLE PRECISION for zherfsx.

Array, **DIMENSION** (*nrhs, n_err_bnds*) . For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3) = 0.0*), then *err_bnds_comp* is not accessed. If *n_err_bnds < 3*, then at most the first $(:, n_err_bnds)$ entries are returned.

The first index in *err_bnds_comp(i, :)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:, err)* contains the following three fields:

<i>err=1</i>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for <i>cherfsx</i> and $\sqrt{n} * dlamch(\epsilon)$ for <i>zherfsx</i> .
<i>err=2</i>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for <i>cherfsx</i> and $\sqrt{n} * dlamch(\epsilon)$ for <i>zherfsx</i> . This error bound should only be trusted if the previous boolean is true.
<i>err=3</i>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for <i>cherfsx</i> and $\sqrt{n} * dlamch(\epsilon)$ for <i>zherfsx</i> to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix <i>Z</i> . Let $z = s^*(a^* \text{diag}(x))$, where <i>x</i> is the solution for the current right-hand side and <i>s</i> scales each row of <i>a^* diag(x)</i> by a power of the radix so all absolute row sums of <i>z</i> are approximately 1.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Output parameter only if the input contains erroneous values, namely, in $\text{params}(1)$, $\text{params}(2)$, $\text{params}(3)$. In such a case, the corresponding elements of params are filled with default values on output.

info INTEGER. If $\text{info} = 0$, the execution is successful. The solution to every right-hand side is guaranteed.

If $\text{info} = -i$, the i -th parameter had an illegal value.

If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $\text{rcond} = 0$ is returned.

If $\text{info} = n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $\text{params}(3) = 0.0$, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that $\text{err_bnds_norm}(j, 1) = 0.0$ or $\text{err_bnds_comp}(j, 1) = 0.0$). See the definition of $\text{err_bnds_norm}(:, 1)$ and $\text{err_bnds_comp}(:, 1)$. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp .

?sprfs

Refines the solution of a system of linear equations with a packed symmetric matrix and estimates the solution error.

Syntax

FORTRAN 77:

```
call ssprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork,
info )
call dsprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, iwork,
info )
call csprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
call zsprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
```

Fortran 95:

```
call sprfs( ap, afp, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_ssprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* ap, const float* afp, const lapack_int* ipiv, const float* b, lapack_int
ldb, float* x, lapack_int ldx, float* ferr, float* berr );
lapack_int LAPACKE_dsprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* ap, const double* afp, const lapack_int* ipiv, const double* b,
lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );
```

```
lapack_int LAPACKE_csprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* ap, const lapack_complex_float* afp, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_zsprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* ap, const lapack_complex_double* afp, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x,
lapack_int ldx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ with a packed symmetric matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?sptrf](#)
- call the solver routine [?ptrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap,afp,b,x,work</i>	REAL for ssprfs DOUBLE PRECISION for dsprfs COMPLEX for csprfs DOUBLE COMPLEX for zsprfs. Arrays: <i>ap</i> (*) contains the original packed matrix A , as supplied to ?sptrf .

afp(*) contains the factored packed matrix *A*, as returned by [?sptrf](#).

b(*ldb*,*) contains the right-hand side matrix *B*.

x(*ldx*,*) contains the solution matrix *X*.

work(*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least $\max(1, n(n+1)/2)$; the second dimension of *b* and *x* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldb INTEGER. The leading dimension of *b*; *ldb* $\geq \max(1, n)$.

ldx INTEGER. The leading dimension of *x*; *ldx* $\geq \max(1, n)$.

ipiv INTEGER.

Array, DIMENSION at least $\max(1, n)$. The *ipiv* array, as returned by [?sptrf](#).

iwork INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork REAL for *csprfs*

DOUBLE PRECISION for *zsprfs*.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

x The refined solution matrix *X*.

ferr, *berr* REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

info INTEGER. If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *sprfs* interface are as follows:

ap Holds the array *A* of size $(n*(n+1)/2)$.

afp Holds the array *AF* of size $(n*(n+1)/2)$.

ipiv Holds the vector of length *n*.

<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be ' <i>U</i> ' or ' <i>L</i> '. The default value is ' <i>U</i> '.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

?hprfs

Refines the solution of a system of linear equations with a packed complex Hermitian matrix and estimates the solution error.

Syntax

FORTRAN 77:

```
call chprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
call zhprfs( uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, ferr, berr, work, rwork,
info )
```

Fortran 95:

```
call hprfs( ap, afp, ipiv, b, x [,uplo] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_chprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* ap, const lapack_complex_float* afp, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* ferr, float* berr );
lapack_int LAPACKE_zhprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* ap, const lapack_complex_double* afp, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x,
lapack_int ldx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ with a packed complex Hermitian matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine:

- call the factorization routine [?hptrf](#)
- call the solver routine [?hptrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap,afp,b,x,work</i>	COMPLEX for chprfs DOUBLE COMPLEX for zhprfs. Arrays: <i>ap</i> (*) contains the original packed matrix A , as supplied to ?hptrf . <i>afp</i> (*) contains the factored packed matrix A , as returned by ?hptrf . <i>b</i> (<i>ldb</i> ,*) contains the right-hand side matrix B . <i>x</i> (<i>ldx</i> ,*) contains the solution matrix X . <i>work</i> (*) is a workspace array. The dimension of arrays <i>ap</i> and <i>afp</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 2*n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; <i>ldb</i> $\geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of <i>x</i> ; <i>ldx</i> $\geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hptrf .

rwork REAL for chprfs
 DOUBLE PRECISION for zhprfs.
 Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	The refined solution matrix <i>X</i> .
<i>ferr</i> , <i>berr</i>	REAL for <i>chprfs</i> . DOUBLE PRECISION for <i>zhprfs</i> .
	Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hprfs` interface are as follows:

<i>ap</i>	Holds the array A of size $(n * (n+1) / 2)$.
<i>afp</i>	Holds the array AF of size $(n * (n+1) / 2)$.
<i>ipiv</i>	Holds the vector of length n .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be ' U ' or ' L '. The default value is ' U '.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $16n^2$ operations. In addition, each step of iterative refinement involves $24n^2$ operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

The real counterpart of this routine is `?ssprfs`/`?dsprfs`.

?trrfs

Estimates the error in the solution of a system of linear equations with a triangular matrix.

Syntax**FORTRAN 77:**

```
call strrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work,
iwork, info )

call dtrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work,
iwork, info )

call ctrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work,
rwork, info )

call ztrrfs( uplo, trans, diag, n, nrhs, a, lda, b, ldb, x, ldx, ferr, berr, work,
rwork, info )
```

Fortran 95:

```
call trrfs( a, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_strrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const float* a, lapack_int lda, const float* b,
lapack_int ldb, const float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_dtrrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const double* a, lapack_int lda, const double* b,
lapack_int ldb, const double* x, lapack_int ldx, double* ferr, double* berr );

lapack_int LAPACKE_ctrrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const lapack_complex_float* a, lapack_int lda,
const lapack_complex_float* b, lapack_int ldb, const lapack_complex_float* x,
lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_ztrrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const lapack_complex_double* a, lapack_int lda,
const lapack_complex_double* b, lapack_int ldb, const lapack_complex_double* x,
lapack_int ldx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the errors in the solution to a system of linear equations $A \cdot X = B$ or $A^T \cdot X = B$ or $A^H \cdot X = B$ with a triangular matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?trtrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', the system has the form $A*X = B$. If <i>trans</i> = 'T', the system has the form $A^T*X = B$. If <i>trans</i> = 'C', the system has the form $A^H*X = B$.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', then A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array <i>a</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a</i> , <i>b</i> , <i>x</i> , <i>work</i>	REAL for strrfs DOUBLE PRECISION for dtrrfs COMPLEX for ctrrfs DOUBLE COMPLEX for ztrrfs. Arrays: <i>a</i> (<i>lda</i> , *) contains the upper or lower triangular matrix A , as specified by <i>uplo</i> . <i>b</i> (<i>ldb</i> , *) contains the right-hand side matrix B . <i>x</i> (<i>ldx</i> , *) contains the solution matrix X . <i>work</i> (*) is a workspace array. The second dimension of <i>a</i> must be at least $\max(1, n)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$.

<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>ctrffs</i> DOUBLE PRECISION for <i>ztrrfs</i> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>ferr</i> , <i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *trrfs* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

?tprrfs

Estimates the error in the solution of a system of linear equations with a packed triangular matrix.

Syntax

FORTRAN 77:

```
call stprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, idx, ferr, berr, work, iwork,
info )
call dtprefs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, idx, ferr, berr, work, iwork,
info )
call ctprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, idx, ferr, berr, work, rwork,
info )
call ztprfs( uplo, trans, diag, n, nrhs, ap, b, ldb, x, idx, ferr, berr, work, rwork,
info )
```

Fortran 95:

```
call tprfs( ap, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_stprfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const float* ap, const float* b, lapack_int ldb,
const float* x, lapack_int idx, float* ferr, float* berr );
lapack_int LAPACKE_dtprefs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const double* ap, const double* b, lapack_int ldb,
const double* x, lapack_int idx, double* ferr, double* berr );
lapack_int LAPACKE_ctprfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const lapack_complex_float* ap,
const lapack_complex_float* b, lapack_int ldb, const lapack_complex_float* x,
lapack_int idx, float* ferr, float* berr );
lapack_int LAPACKE_ztprfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int nrhs, const lapack_complex_double* ap,
const lapack_complex_double* b, lapack_int ldb, const lapack_complex_double* x,
lapack_int idx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the errors in the solution to a system of linear equations $A \cdot X = B$ or $A^T \cdot X = B$ or $A^H \cdot X = B$ with a packed triangular matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?ptrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether A is upper or lower triangular: If $uplo = 'U'$, then A is upper triangular. If $uplo = 'L'$, then A is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'.
	Indicates the form of the equations: If $trans = 'N'$, the system has the form $A \cdot X = B$. If $trans = 'T'$, the system has the form $A^T \cdot X = B$. If $trans = 'C'$, the system has the form $A^H \cdot X = B$.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'.
	If $diag = 'N'$, A is not a unit triangular matrix. If $diag = 'U'$, A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array <i>ap</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ap, b, x, work</i>	REAL for <i>stprfs</i> DOUBLE PRECISION for <i>dtprrfs</i> COMPLEX for <i>ctprrf</i> DOUBLE COMPLEX for <i>ztprrf</i> . Arrays: <i>ap(*)</i> contains the upper or lower triangular matrix A , as specified by <i>uplo</i> . <i>b(ldb, *)</i> contains the right-hand side matrix B . <i>x(lidx, *)</i> contains the solution matrix X . <i>work(*)</i> is a workspace array. The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$; the dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>lidx</i>	INTEGER. The leading dimension of <i>x</i> ; $lidx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork REAL for ctprfs
 DOUBLE PRECISION for ztprfs.
 Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>ferr, berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, <i>nrhs</i>). Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tprfs` interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n^* (n+1) / 2)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in `ferr` are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors or $4n^2$ for complex flavors.

?tbrfs

Estimates the error in the solution of a system of linear equations with a triangular band matrix.

Syntax

FORTRAN 77:

```
call stbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr,
work, iwork, info )

call dtbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr,
work, iwork, info )

call ctbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr,
work, rwork, info )

call ztbrfs( uplo, trans, diag, n, kd, nrhs, ab, ldab, b, ldb, x, ldx, ferr, berr,
work, rwork, info )
```

Fortran 95:

```
call tbrfs( ab, b, x [,uplo] [,trans] [,diag] [,ferr] [,berr] [,info] )
```

C:

```
lapack_int LAPACKE_stbrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int kd, lapack_int nrhs, const float* ab, lapack_int ldab,
const float* b, lapack_int ldb, const float* x, lapack_int ldx, float* ferr, float*
berr );

lapack_int LAPACKE_dtbrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int kd, lapack_int nrhs, const double* ab, lapack_int ldab,
const double* b, lapack_int ldb, const double* x, lapack_int ldx, double* ferr,
double* berr );

lapack_int LAPACKE_ctbrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int kd, lapack_int nrhs, const lapack_complex_float* ab,
lapack_int ldab, const lapack_complex_float* b, lapack_int ldb,
const lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_ztbrfs( int matrix_layout, char uplo, char trans, char diag,
lapack_int n, lapack_int kd, lapack_int nrhs, const lapack_complex_double* ab,
lapack_int ldab, const lapack_complex_double* b, lapack_int ldb,
const lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates the errors in the solution to a system of linear equations $A \cdot X = B$ or $A^T \cdot X = B$ or $A^H \cdot X = B$ with a triangular band matrix A , with multiple right-hand sides. For each computed solution vector x , the routine computes the *component-wise backward error* β . This error is the smallest relative perturbation in elements of A and b such that x is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \text{ such that } (A + \delta A)x = (b + \delta b).$$

The routine also estimates the *component-wise forward error* in the computed solution $\|x - x_e\|_\infty / \|x\|_\infty$ (here x_e is the exact solution).

Before calling this routine, call the solver routine [?tbtrs](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If $uplo = 'U'$, then A is upper triangular. If $uplo = 'L'$, then A is lower triangular.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If $trans = 'N'$, the system has the form $A \cdot X = B$. If $trans = 'T'$, the system has the form $A^T \cdot X = B$. If $trans = 'C'$, the system has the form $A^H \cdot X = B$.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If $diag = 'N'$, A is not a unit triangular matrix. If $diag = 'U'$, A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array <i>ab</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of super-diagonals or sub-diagonals in the matrix A ; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>ab, b, x, work</i>	REAL for stbrfs DOUBLE PRECISION for dtbrfs COMPLEX for ctbrfs DOUBLE COMPLEX for ztbrfs. Arrays: <i>ab</i> (<i>ldab</i> , *) contains the upper or lower triangular matrix A , as specified by <i>uplo</i> , in band storage format. <i>b</i> (<i>ldb</i> , *) contains the right-hand side matrix B . <i>x</i> (<i>ldx</i> , *) contains the solution matrix X . <i>work</i> (*) is a workspace array. The second dimension of <i>a</i> must be at least $\max(1, n)$; the second dimension of <i>b</i> and <i>x</i> must be at least $\max(1, nrhs)$. The dimension of <i>work</i> must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; $ldab \geq kd + 1$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of <i>x</i> ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

rwork REAL for ctbrfs
 DOUBLE PRECISION for ztbrfs.
 Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>ferr, berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, <i>nrhs</i>). Contain the component-wise forward and backward errors, respectively, for each solution vector.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tbrfs` interface are as follows:

<i>ab</i>	Holds the array <i>A</i> of size $(kd+1, n)$.
<i>b</i>	Holds the matrix <i>B</i> of size $(n, nrhs)$.
<i>x</i>	Holds the matrix <i>X</i> of size $(n, nrhs)$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>diag</i>	Must be 'N' or 'U'. The default value is 'N'.

Application Notes

The bounds returned in `ferr` are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n*kd$ floating-point operations for real flavors or $8n*kd$ operations for complex flavors.

Routines for Matrix Inversion

It is seldom necessary to compute an explicit inverse of a matrix. In particular, do not attempt to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see [Routines for Solving Systems of Linear Equations](#)); this is more efficient and more accurate.

However, matrix inversion routines are provided for the rare occasions when an explicit inverse matrix is needed.

?getri

Computes the inverse of an LU-factored general matrix.

Syntax

FORTRAN 77:

```
call sgetri( n, a, lda, ipiv, work, lwork, info )
call dgetri( n, a, lda, ipiv, work, lwork, info )
call cgetri( n, a, lda, ipiv, work, lwork, info )
call zgetri( n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call getri( a, ipiv [,info] )
```

C:

```
lapack_int LAPACKE_<?>getri( int matrix_layout, lapack_int n, <datatype>* a, lapack_int
lda, const lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a general matrix A . Before calling this routine, call [?getrf](#) to factorize A .

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a, work</i>	REAL for sgetri DOUBLE PRECISION for dgetri COMPLEX for cgetri DOUBLE COMPLEX for zgetri. Arrays: $a(\text{lda}, *)$, $work(*)$. $a(\text{lda}, *)$ contains the factorization of the matrix A , as returned by ?getrf : $A = P \cdot L \cdot U$. The second dimension of a must be at least $\max(1, n)$. $work(*)$ is a workspace array of dimension at least $\max(1, lwork)$.

<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?getrf .
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; $lwork \geq n$. If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> .

See *Application Notes* below for the suggested value of *lwork*.

Output Parameters

<i>a</i>	Overwritten by the n -by- n matrix $\text{inv}(A)$.
<i>work(1)</i>	If $info = 0$, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the <i>i</i> -th parameter had an illegal value. If $info = i$, the <i>i</i> -th diagonal element of the factor <i>U</i> is zero, <i>U</i> is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *getri* interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (n, n) .
<i>ipiv</i>	Holds the vector of length n .

Application Notes

For better performance, try using $lwork = n * \text{blocksize}$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed inverse X satisfies the following error bound:

$$|XA - I| \leq c(n)\varepsilon|X||P||L||U|,$$

where $c(n)$ is a modest linear function of n ; ε is the machine precision; I denotes the identity matrix; P , L , and U are the factors of the matrix factorization $A = P^*L^*U$.

The total number of floating-point operations is approximately $(4/3)n^3$ for real flavors and $(16/3)n^3$ for complex flavors.

?potri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix.

Syntax

FORTRAN 77:

```
call spotri( uplo, n, a, lda, info )
call dpotri( uplo, n, a, lda, info )
call cpotri( uplo, n, a, lda, info )
call zpotri( uplo, n, a, lda, info )
```

Fortran 95:

```
call potri( a [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>potri( int matrix_layout, char uplo, lapack_int n, <datatype>* a,
lapack_int lda );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix A . Before calling this routine, call [?potrf](#) to factorize A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether A is upper or lower triangular:
	If <i>uplo</i> = 'U', then A is upper triangular.
	If <i>uplo</i> = 'L', then A is lower triangular.

n INTEGER. The order of the matrix A ; $n \geq 0$.

a REAL for spotri
 DOUBLE PRECISION for dpotri
 COMPLEX for cpotri
 DOUBLE COMPLEX for zpotri.

Array *a*(*lda*, *). Contains the factorization of the matrix *A*, as returned by [?potrf](#).

The second dimension of *a* must be at least $\max(1, n)$.

Output Parameters

a Overwritten by the n -by- n matrix $\text{inv}(A)$.

info INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `potri` interface are as follows:

a Holds the matrix A of size (n, n) .

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$\|XA - I\|_2 \leq c(n)\kappa_2(A), \quad \|AX - I\|_2 \leq c(n)\kappa_2(A),$$

where $c(n)$ is a modest linear function of n , and ε is the machine precision; I denotes the identity matrix.

The 2-norm $\|A\|_2$ of a matrix A is defined by $\|A\|_2 = \max_{x: x \neq 0} (Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$.

The total number of floating-point operations is approximately $(2/3) n^3$ for real flavors and $(8/3) n^3$ for complex flavors.

?pftri

Computes the inverse of a symmetric (Hermitian) positive-definite matrix in RFP format using the Cholesky factorization.

Syntax

FORTRAN 77:

```
call spftri( transr, uplo, n, a, info )
call dpftri( transr, uplo, n, a, info )
call cpftri( transr, uplo, n, a, info )
call zpftri( transr, uplo, n, a, info )
```

C:

```
lapack_int LAPACKE_<?>pftri( int matrix_layout, char transr, char uplo, lapack_int n,
<datatype>* a );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric positive definite or, for complex data, Hermitian positive-definite matrix A using the Cholesky factorization:

$$\begin{aligned} A &= U^T * U \text{ for real data, } A = U^H * U \text{ for complex data} && \text{if } uplo='U' \\ A &= L * L^T \text{ for real data, } A = L * L^H \text{ for complex data} && \text{if } uplo='L' \end{aligned}$$

Before calling this routine, call [?pftrf](#) to factorize A .

The matrix A is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>transr</i>	CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data).
	If <i>transr</i> = 'N', the Normal <i>transr</i> of RFP A is stored.
	If <i>transr</i> = 'T', the Transpose <i>transr</i> of RFP A is stored.
	If <i>transr</i> = 'C', the Conjugate-Transpose <i>transr</i> of RFP A is stored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of the RFP matrix A is stored:
	If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A .
	If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i>	REAL for spftri DOUBLE PRECISION for dpftri

COMPLEX for `cpftri`

DOUBLE COMPLEX for `zpftri`.

Array, DIMENSION ($n * (n+1) / 2$). The array `a` contains the matrix A in the RFP format.

Output Parameters

`a` The symmetric/Hermitian inverse of the original matrix in the same storage format.

`info` INTEGER. If `info=0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

If `info = i`, the (i,i) element of the factor `U` or `L` is zero, and the inverse could not be computed.

?pptri

Computes the inverse of a packed symmetric (Hermitian) positive-definite matrix

Syntax

FORTRAN 77:

```
call spptri( uplo, n, ap, info )
call dpptri( uplo, n, ap, info )
call cpptri( uplo, n, ap, info )
call zpptri( uplo, n, ap, info )
```

Fortran 95:

```
call pptri( ap [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>pptri( int matrix_layout, char uplo, lapack_int n, <datatype>* ap );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine computes the inverse `inv(A)` of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix A in *packed* form. Before calling this routine, call `?ptrf` to factorize A .

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

`uplo` CHARACTER*1. Must be '`U`' or '`L`'.

Indicates whether the upper or lower triangular factor is stored in *ap*:

If *uplo* = 'U', then the upper triangular factor is stored.

If *uplo* = 'L', then the lower triangular factor is stored.

n

INTEGER. The order of the matrix *A*; $n \geq 0$.

ap

REAL for *spptri*

DOUBLE PRECISION for *dpptri*

COMPLEX for *cpptri*

DOUBLE COMPLEX for *zpptri*.

Array, DIMENSION at least $\max(1, n(n+1)/2)$.

Contains the factorization of the packed matrix *A*, as returned by [?pptrf](#).

The dimension *ap* must be at least $\max(1, n(n+1)/2)$.

Output Parameters

ap

Overwritten by the packed *n*-by-*n* matrix *inv(A)*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *pptri* interface are as follows:

ap

Holds the array *A* of size $(n^*(n+1)/2)$.

uplo

Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$\|XA - I\|_2 \leq c(n)\varepsilon\kappa_2(A), \quad \|AX - I\|_2 \leq c(n)\varepsilon\kappa_2(A),$$

where *c(n)* is a modest linear function of *n*, and ε is the machine precision; *I* denotes the identity matrix.

The 2-norm $\|A\|_2$ of a matrix *A* is defined by $\|A\|_2 = \max_{x \neq 0} (Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?sytri

Computes the inverse of a symmetric matrix.

Syntax**FORTRAN 77:**

```
call ssytri( uplo, n, a, lda, ipiv, work, info )
call dsytri( uplo, n, a, lda, ipiv, work, info )
call csytri( uplo, n, a, lda, ipiv, work, info )
call zsytri( uplo, n, a, lda, ipiv, work, info )
```

Fortran 95:

```
call sytri( a, ipiv [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>sytri( int matrix_layout, char uplo, lapack_int n, <datatype>* a,
lapack_int lda, const lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric matrix A . Before calling this routine, call [?sytrf](#) to factorize A .

Input Parameters

The data types are given for the Fortran interface. A $<\text{datatype}>$ placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array a stores the Bunch-Kaufman factorization $A = P \cdot U \cdot D \cdot U^T \cdot P^T$. If $uplo = 'L'$, the array a stores the Bunch-Kaufman factorization $A = P \cdot L \cdot D \cdot L^T \cdot P^T$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a, work</i>	REAL for ssytri DOUBLE PRECISION for dsytri COMPLEX for csytri DOUBLE COMPLEX for zsytri. Arrays: $a(1:da, :)$ contains the factorization of the matrix A , as returned by ?sytrf .

The second dimension of a must be at least $\max(1, n)$.

$work(*)$ is a workspace array.

The dimension of $work$ must be at least $\max(1, 2*n)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

$ipiv$ INTEGER.

Array, DIMENSION at least $\max(1, n)$.

The $ipiv$ array, as returned by [?sytrf](#).

Output Parameters

a Overwritten by the n -by- n matrix $\text{inv}(A)$.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of D is zero, D is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytri` interface are as follows:

a Holds the matrix A of size (n, n) .

$ipiv$ Holds the vector of length n .

$uplo$ Must be '`U`' or '`L`'. The default value is '`U`'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|D^T P^T X P U - I| \leq c(n) \epsilon (|D| |U^T P^T X P| U + |D| |D^{-1}|)$$

for $uplo = 'U'$, and

$$|D L^T P^T X P L - I| \leq c(n) \epsilon (|D| |L^T P^T X P| L + |D| |D^{-1}|)$$

for $uplo = 'L'$. Here $c(n)$ is a modest linear function of n , and ϵ is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?sytri_rook

Computes the inverse of a symmetric matrix using $U^T D U$ or $L^T D L$ bounded Bunch-Kaufman factorization.

Syntax

FORTRAN 77:

```
call ssytri_rook( uplo, n, a, lda, ipiv, work, info )
call dsytri_rook( uplo, n, a, lda, ipiv, work, info )
call csytri_rook( uplo, n, a, lda, ipiv, work, info )
call zsytri_rook( uplo, n, a, lda, ipiv, work, info )
```

Fortran 95:

```
call sytri_rook( a, ipiv [,uplo] [,info] )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric matrix A . Before calling this routine, call [?sytrf_rook](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the factorization $A = U*D*U^T$. If <i>uplo</i> = 'L', the array <i>a</i> stores the factorization $A = L*D*L^T$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i> , <i>work</i>	REAL for ssytri_rook DOUBLE PRECISION for dsytri_rook COMPLEX for csytri_rook DOUBLE COMPLEX for zsytri_rook. Arrays: <i>a</i> (size <i>lda</i> by *) contains the factorization of the matrix A , as returned by ?sytrf_rook . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?sytrf_rook .

Output Parameters

<i>a</i>	Overwritten by the n -by- n matrix $\text{inv}(A)$. If $\text{uplo} = \text{'U'}$, the upper triangular part of the inverse is formed and the part of <i>a</i> below the diagonal is not referenced; if $\text{uplo} = \text{'L'}$ the lower triangular part of the inverse is formed and the part of <i>a</i> above the diagonal is not referenced."
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of <i>D</i> is zero, <i>D</i> is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytri_rook` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

See Also

[Matrix Storage Schemes](#)

?hetri

Computes the inverse of a complex Hermitian matrix.

Syntax

FORTRAN 77:

```
call chetri( uplo, n, a, lda, ipiv, work, info )
call zhetri( uplo, n, a, lda, ipiv, work, info )
```

Fortran 95:

```
call hetri( a, ipiv [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>hetri( int matrix_layout, char uplo, lapack_int n, <datatype>* a,
lapack_int lda, const lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi

- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a complex Hermitian matrix A . Before calling this routine, call [?hetrf](#) to factorize A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If <i>uplo</i> = 'U', the array <i>a</i> stores the Bunch-Kaufman factorization $A = P*U*D*U^H*P^T$. If <i>uplo</i> = 'L', the array <i>a</i> stores the Bunch-Kaufman factorization $A = P*L*D*L^H*P^T$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i> , <i>work</i>	COMPLEX for chetri DOUBLE COMPLEX for zhetri. Arrays: <i>a</i> (<i>lda</i> , *) contains the factorization of the matrix A , as returned by ?hetrf . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $\text{lda} \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The <i>ipiv</i> array, as returned by ?hetrf .

Output Parameters

<i>a</i>	Overwritten by the n -by- n matrix $\text{inv}(A)$.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value. If <i>info</i> = i , the <i>i</i> -th diagonal element of <i>D</i> is zero, <i>D</i> is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetri` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|D^*U^H P^T X^*P^*U - I| \leq c(n)\varepsilon(|D||U^H|P^T|X|P|U| + |D||D^{-1}|)$$

for $uplo = 'U'$, and

$$|D^*L^H P^T X^*P^*L - I| \leq c(n)\varepsilon(|D||L^H|P^T|X|P|L| + |D||D^{-1}|)$$

for $uplo = 'L'$. Here $c(n)$ is a modest linear function of n , and ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is [?sytri](#).

?hetri_rook

Computes the inverse of a complex Hermitian matrix using $U^*D^*U^H$ or $L^*D^*L^H$ bounded Bunch-Kaufman factorization.

Syntax

FORTRAN 77:

```
call chetri_rook( uplo, n, a, lda, ipiv, work, info )
call zhetri_rook( uplo, n, a, lda, ipiv, work, info )
```

Fortran 95:

```
call hetri_rook( a, ipiv [,uplo] [,info] )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a complex Hermitian matrix A . Before calling this routine, call [?hetrf_rook](#) to factorize A .

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array a stores the factorization $A = U*D*U^H$. If $uplo = 'L'$, the array a stores the factorization $A = L*D*L^H$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a, work</i>	COMPLEX for <code>chetri_rook</code> DOUBLE COMPLEX for <code>zhetri_rook</code> . Arrays: $a(1:da, :)$ contains the factorization of the matrix A , as returned by ?hetrf_rook . The second dimension of a must be at least $\max(1, n)$. $work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, size at least $\max(1, n)$. The $ipiv$ array, as returned by ?hetrf_rook .

Output Parameters

<i>a</i>	Overwritten by the n -by- n matrix $\text{inv}(A)$.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, the i -th diagonal element of D is zero, D is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetri_rook` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The total number of floating-point operations is approximately $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is [?sytri_rook](#).

See Also

[Matrix Storage Schemes](#)

?sytri2

Computes the inverse of a symmetric indefinite matrix through setting the leading dimension of the workspace and calling ?sytri2x.

Syntax

FORTRAN 77:

```
call ssytri2( uplo, n, a, lda, ipiv, work, lwork, info )
call dsytri2( uplo, n, a, lda, ipiv, work, lwork, info )
call csytri2( uplo, n, a, lda, ipiv, work, lwork, info )
call zsytri2( uplo, n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call sytri2( a, ipiv[, uplo] [, info] )
```

C:

```
lapack_int LAPACKE_<?>sytri2( int matrix_layout, char uplo, lapack_int n, <datatype>* a, lapack_int lda, const lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a symmetric indefinite matrix A using the factorization $A = U*D*U^T$ or $A = L*D*L^T$ computed by [?sytrf](#).

The [?sytri2](#) routine sets the leading dimension of the workspace before calling [?sytri2x](#) that actually computes the inverse.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>matrix_layout</code>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array a stores the factorization $A = U*D*U^T$.

If $uplo = 'L'$, the array a stores the factorization $A = L \cdot D \cdot L^T$.

n INTEGER. The order of the matrix A ; $n \geq 0$.

$a, work$

- REAL for ssytri2
- DOUBLE PRECISION for dsytri2
- COMPLEX for csytri2
- DOUBLE COMPLEX for zsytri2

Array a (size lda by n) contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as returned by ?sytrf.

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array of $(n+nb+1) * (nb+3)$ dimension.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

$ipiv$ INTEGER.

Array, size at least $\max(1, n)$.

Details of the interchanges and the block structure of D as returned by ?sytrf.

$lwork$ INTEGER. The dimension of the $work$ array.

$lwork \geq (n+nb+1) * (nb+3)$

where

nb is the block size parameter as returned by sytrf.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla.

Output Parameters

a

If $info = 0$, the symmetric inverse of the original matrix.

If $uplo = 'U'$, the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced.

If $uplo = 'L'$, the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, $D(i,i) = 0$; D is singular and its inversion could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytri2` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Indicates how the matrix A has been factored. Must be ' <code>U</code> ' or ' <code>L</code> '.

See Also

[?sytrf](#)

[?sytri2x](#)

[Matrix Storage Schemes](#)

?hetri2

Computes the inverse of a Hermitian indefinite matrix through setting the leading dimension of the workspace and calling ?hetri2x.

Syntax

FORTRAN 77:

```
call chetri2( uplo, n, a, lda, ipiv, work, lwork, info )
call zhetri2( uplo, n, a, lda, ipiv, work, lwork, info )
```

Fortran 95:

```
call hetri2( a, ipiv[, uplo][, info] )
```

C:

```
lapack_int LAPACKE_<?>hetri2( int matrix_layout, char uplo, lapack_int n, <datatype>* a, lapack_int lda, const lapack_int* ipiv );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine computes the inverse $\text{inv}(A)$ of a Hermitian indefinite matrix A using the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ computed by [?hetrf](#).

The `?hetri2` routine sets the leading dimension of the workspace before calling `?hetri2x` that actually computes the inverse.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '.
-------------------	--

Indicates how the input matrix A has been factored:

If $uplo = 'U'$, the array a stores the factorization $A = U^*D^*U^H$.

If $uplo = 'L'$, the array a stores the factorization $A = L^*D^*L^H$.

n

INTEGER. The order of the matrix A ; $n \geq 0$.

$a, work$

COMPLEX for chetri2

DOUBLE COMPLEX for zhetri2

Arrays:

$a(1:da, :)$ contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as returned by ?sytrf.

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array of $(n+nb+1) * (nb+3)$ dimension.

lda

INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

$ipiv$

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

Details of the interchanges and the block structure of D as returned by ?hetrf.

$lwork$

INTEGER. The dimension of the $work$ array.

$lwork \geq (n+nb+1) * (nb+3)$

where

nb is the block size parameter as returned by hetrf.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla.

Output Parameters

a

If $info = 0$, the inverse of the original matrix.

If $info = 'U'$, the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced.

If $info = 'L'$, the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, $D(i,i) = 0$; D is singular and its inversion could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetri2` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>uplo</code>	Indicates how the input matrix A has been factored. Must be ' <code>U</code> ' or ' <code>L</code> '.

See Also

[?hetrf](#)
[?hetri2x](#)

?sytri2x

Computes the inverse of a symmetric indefinite matrix after ?sytri2 sets the leading dimension of the workspace.

Syntax

FORTRAN 77:

```
call ssytri2x( uplo, n, a, lda, ipiv, work, nb, info )
call dsytri2x( uplo, n, a, lda, ipiv, work, nb, info )
call csytri2x( uplo, n, a, lda, ipiv, work, nb, info )
call zsytri2x( uplo, n, a, lda, ipiv, work, nb, info )
```

Fortran 95:

```
call sytri2x( a, ipiv, nb[, uplo][, info] )
```

C:

```
lapack_int LAPACKE_<?>sytri2x( int matrix_layout, char uplo, lapack_int n, <datatype>* a, lapack_int lda, const lapack_int* ipiv, lapack_int nb );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine computes the inverse `inv(A)` of a symmetric indefinite matrix A using the factorization $A = U^*D^*U^T$ or $A = L^*D^*L^T$ computed by [?sytrf](#).

The `?sytri2x` actually computes the inverse after the `?sytri2` routine sets the leading dimension of the workspace before calling `?sytri2x`.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If $uplo = 'U'$, the array a stores the factorization $A = U^*D^*U^T$.
	If $uplo = 'L'$, the array a stores the factorization $A = L^*D^*L^T$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a, work</i>	REAL for ssytri2x DOUBLE PRECISION for dsytri2x COMPLEX for csytri2x DOUBLE COMPLEX for zsytri2x
	Arrays: $a(1da,*)$ contains the nb (block size) diagonal matrix D and the multipliers used to obtain the factor U or L as returned by ?sytrf. The second dimension of a must be at least $\max(1, n)$. $work$ is a workspace array of the dimension $(n+nb+1) * (nb+3)$ where nb is the block size as set by ?sytrf.
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. Details of the interchanges and the nb structure of D as returned by ?sytrf.
<i>nb</i>	INTEGER. Block size.

Output Parameters

<i>a</i>	If $info = 0$, the symmetric inverse of the original matrix. If $info = 'U'$, the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced. If $info = 'L'$, the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, $D_{ii} = 0$; D is singular and its inversion could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytri2x` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>nb</code>	Holds the block size.
<code>uplo</code>	Indicates how the input matrix A has been factored. Must be ' <code>U</code> ' or ' <code>L</code> '.

See Also

[?sytrf](#)
[?sytri2](#)

?hetri2x

Computes the inverse of a Hermitian indefinite matrix after ?hetri2 sets the leading dimension of the workspace.

Syntax

FORTRAN 77:

```
call chetri2x( uplo, n, a, lda, ipiv, work, nb, info )
call zhetri2x( uplo, n, a, lda, ipiv, work, nb, info )
```

Fortran 95:

```
call hetri2x( a, ipiv, nb[, uplo][, info] )
```

C:

```
lapack_int LAPACKE_<?>hetri2x( int matrix_layout, char uplo, lapack_int n, <datatype>* a, lapack_int lda, const lapack_int* ipiv, lapack_int nb );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine computes the inverse $\text{inv}(A)$ of a Hermitian indefinite matrix A using the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ computed by `?hetrf`.

The `?hetri2x` actually computes the inverse after the `?hetri2` routine sets the leading dimension of the workspace before calling `?hetri2x`.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If $uplo = 'U'$, the array a stores the factorization $A = U^*D^*U^H$.
	If $uplo = 'L'$, the array a stores the factorization $A = L^*D^*L^H$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a, work</i>	COMPLEX for chetri2x DOUBLE COMPLEX for zhetri2x
	Arrays: $a(1:da, :)$ contains the nb (block size) diagonal matrix D and the multipliers used to obtain the factor U or L as returned by ?hetrf. The second dimension of a must be at least $\max(1, n)$. $work$ is a workspace array of the dimension $(n+nb+1) * (nb+3)$ where nb is the block size as set by ?hetrf.
<i>lda</i>	INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. Details of the interchanges and the nb structure of D as returned by ?hetrf.
<i>nb</i>	INTEGER. Block size.

Output Parameters

<i>a</i>	If $info = 0$, the symmetric inverse of the original matrix. If $info = 'U'$, the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced. If $info = 'L'$, the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, $D_{ii} = 0$; D is singular and its inversion could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetri2x` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>ipiv</i>	Holds the vector of length n .
<i>nb</i>	Holds the block size.
<i>uplo</i>	Indicates how the input matrix A has been factored. Must be ' U ' or ' L '.

See Also

?hetrf

?hetri2

?sptri

Computes the inverse of a symmetric matrix using packed storage.

Syntax

FORTRAN 77:

```
call ssptri( uplo, n, ap, ipiv, work, info )
call dsptri( uplo, n, ap, ipiv, work, info )
call csptri( uplo, n, ap, ipiv, work, info )
call zsptri( uplo, n, ap, ipiv, work, info )
```

Fortran 95:

```
call sptri( ap, ipiv [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>sptri( int matrix_layout, char uplo, lapack_int n, <datatype>* ap, const lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a packed symmetric matrix A . Before calling this routine, call [?sptrf](#) to factorize A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be ' U ' or ' L '.
	Indicates how the input matrix A has been factored:
	If $uplo = 'U'$, the array <i>ap</i> stores the Bunch-Kaufman factorization $A = P \cdot U \cdot D \cdot U^T \cdot P^T$.

If $uplo = 'L'$, the array ap stores the Bunch-Kaufman factorization $A = P^*L^*D^*L^T*P^T$.

n INTEGER. The order of the matrix A ; $n \geq 0$.

$ap, work$

REAL for ssptri
 DOUBLE PRECISION for dsptri
 COMPLEX for csptri
 DOUBLE COMPLEX for zsptri.

Arrays:

$ap(*)$ contains the factorization of the matrix A , as returned by [?sptrf](#).

The dimension of ap must be at least $\max(1, n(n+1)/2)$.

$work(*)$ is a workspace array.

The dimension of $work$ must be at least $\max(1, n)$.

$ipiv$ INTEGER.

Array, DIMENSION at least $\max(1, n)$. The $ipiv$ array, as returned by [?sptrf](#).

Output Parameters

ap Overwritten by the n -by- n matrix $\text{inv}(A)$ in packed form.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of D is zero, D is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sptri` interface are as follows:

ap Holds the array A of size $(n*(n+1)/2)$.

$ipiv$ Holds the vector of length n .

$uplo$ Must be '`U`' or '`L`'. The default value is '`U`'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|D^*U^T*P^T*X*P*U - I| \leq c(n)\epsilon(|D| |U^T| |P^T| |X| |P| |U| + |D| |D^{-1}|)$$

for $uplo = 'U'$, and

$$|D^*L^T*P^T*X*P*L - I| \leq c(n)\varepsilon(|D||L^T|P^T|X|P|L| + |D||D^{-1}|)$$

for $uplo = 'L'$. Here $c(n)$ is a modest linear function of n , and ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

?hptri

Computes the inverse of a complex Hermitian matrix using packed storage.

Syntax

FORTRAN 77:

```
call chptri( uplo, n, ap, ipiv, work, info )
call zhptri( uplo, n, ap, ipiv, work, info )
```

Fortran 95:

```
call hptri( ap, ipiv [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>hptri( int matrix_layout, char uplo, lapack_int n, <datatype>* ap, const lapack_int* ipiv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a complex Hermitian matrix A using packed storage. Before calling this routine, call [?hptrf](#) to factorize A .

Input Parameters

The data types are given for the Fortran interface. A $<\text{datatype}>$ placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If $uplo = 'U'$, the array <i>ap</i> stores the packed Bunch-Kaufman factorization $A = P^*U*D^*U^H*P^T$.
	If $uplo = 'L'$, the array <i>ap</i> stores the packed Bunch-Kaufman factorization $A = P^*L*D^*L^H*P^T$.
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>ap, work</i>	COMPLEX for chptri DOUBLE COMPLEX for zhptri.

Arrays:

ap(*) contains the factorization of the matrix *A*, as returned by [?hptrf](#).

The dimension of *ap* must be at least $\max(1, n(n+1)/2)$.

work(*) is a workspace array.

The dimension of *work* must be at least $\max(1, n)$.

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

The *ipiv* array, as returned by [?hptrf](#).

Output Parameters

ap

Overwritten by the *n*-by-*n* matrix $\text{inv}(A)$.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *D* is zero, *D* is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hptri` interface are as follows:

ap

Holds the array *A* of size $(n^*(n+1)/2)$.

ipiv

Holds the vector of length *n*.

uplo

Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed inverse *X* satisfies the following error bounds:

$$|D^*U^H P^T X P U - I| \leq c(n) \varepsilon (|D| |U^H| P^T |X| P |U| + |D| |D^{-1}|)$$

for *uplo* = 'U', and

$$|D^*L^H P^T X P L - I| \leq c(n) \varepsilon (|D| |L^H| P^T |X| P |L| + |D| |D^{-1}|)$$

for *uplo* = 'L'. Here *c(n)* is a modest linear function of *n*, and *ε* is the machine precision; *I* denotes the identity matrix.

The total number of floating-point operations is approximately $(2/3)n^3$ for real flavors and $(8/3)n^3$ for complex flavors.

The real counterpart of this routine is [?sptri](#).

?trtri

Computes the inverse of a triangular matrix.

Syntax**FORTRAN 77:**

```
call strtri( uplo, diag, n, a, lda, info )
call dtrtri( uplo, diag, n, a, lda, info )
call ctrtri( uplo, diag, n, a, lda, info )
call ztrtri( uplo, diag, n, a, lda, info )
```

Fortran 95:

```
call trtri( a [,uplo] [,diag] [,info] )
```

C:

```
lapack_int LAPACKE_<?>trtri( int matrix_layout, char uplo, char diag, lapack_int n,
<datatype>* a, lapack_int lda );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a triangular matrix A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If <i>uplo</i> = 'U', then A is upper triangular. If <i>uplo</i> = 'L', then A is lower triangular.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>diag</i> = 'N', then A is not a unit triangular matrix. If <i>diag</i> = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array <i>a</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i>	REAL for strtri DOUBLE PRECISION for dtrtri COMPLEX for ctrtri DOUBLE COMPLEX for ztrtri. Array: DIMENSION (, *).

Contains the matrix A .

The second dimension of a must be at least $\max(1, n)$.

lda

INTEGER. The first dimension of a ; $lda \geq \max(1, n)$.

Output Parameters

a

Overwritten by the n -by- n matrix $\text{inv}(A)$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of A is zero, A is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trtri` interface are as follows:

a

Holds the matrix A of size (n, n) .

$uplo$

Must be '`U`' or '`L`'. The default value is '`U`'.

$diag$

Must be '`N`' or '`U`'. The default value is '`N`'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|XA - I| \leq c(n)\varepsilon |X| |A|$$

$$|XA - I| \leq c(n)\varepsilon |A^{-1}| |A| |X|,$$

where $c(n)$ is a modest linear function of n ; ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

?tftri

Computes the inverse of a triangular matrix stored in the Rectangular Full Packed (RFP) format.

Syntax

FORTRAN 77:

```
call stftri( transr, uplo, diag, n, a, info )
call dtftri( transr, uplo, diag, n, a, info )
call ctftri( transr, uplo, diag, n, a, info )
call ztftri( transr, uplo, diag, n, a, info )
```

C:

```
lapack_int LAPACKE_<?>tftri( int matrix_layout, char transr, char uplo, char diag,
lapack_int n, <datatype>* a );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

Computes the inverse of a triangular matrix A stored in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

This is the block version of the algorithm, calling Level 3 BLAS.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>transr</i>	CHARACTER*1. Must be 'N', 'T' (for real data) or 'C' (for complex data).
	If <i>transr</i> = 'N', the Normal <i>transr</i> of RFP A is stored.
	If <i>transr</i> = 'T', the Transpose <i>transr</i> of RFP A is stored.
	If <i>transr</i> = 'C', the Conjugate-Transpose <i>transr</i> of RFP A is stored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of RFP A is stored:
	If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix A .
	If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix A .
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'.
	If <i>diag</i> = 'N', then A is not a unit triangular matrix.
	If <i>diag</i> = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array <i>a</i> .
<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i>	REAL for stftri DOUBLE PRECISION for dtftri COMPLEX for ctftri DOUBLE COMPLEX for ztftri. Array, DIMENSION ($n * (n+1) / 2$). The array <i>a</i> contains the matrix A in the RFP format.

Output Parameters

<i>a</i>	The (triangular) inverse of the original matrix in the same storage format.
<i>info</i>	INTEGER. If $info=0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, $A(i,i)$ is exactly zero. The triangular matrix is singular and its inverse cannot be computed.

?tptri

Computes the inverse of a triangular matrix using packed storage.

Syntax

FORTRAN 77:

```
call stptri( uplo, diag, n, ap, info )
call dtptri( uplo, diag, n, ap, info )
call ctptri( uplo, diag, n, ap, info )
call ztptri( uplo, diag, n, ap, info )
```

Fortran 95:

```
call tptri( ap [,uplo] [,diag] [,info] )
```

C:

```
lapack_int LAPACKE_<?>tptri( int matrix_layout, char uplo, char diag, lapack_int n,
<datatype>* ap );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a packed triangular matrix A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether A is upper or lower triangular: If $uplo = 'U'$, then A is upper triangular. If $uplo = 'L'$, then A is lower triangular.
<i>diag</i>	CHARACTER*1. Must be 'N' or 'U'.

If $diag = 'N'$, then A is not a unit triangular matrix.

If $diag = 'U'$, A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array ap .

n

INTEGER. The order of the matrix A ; $n \geq 0$.

ap

REAL for stptri

DOUBLE PRECISION for dtptri

COMPLEX for ctptri

DOUBLE COMPLEX for ztptri.

Array, DIMENSION at least $\max(1, n(n+1)/2)$.

Contains the packed triangular matrix A .

Output Parameters

ap

Overwritten by the packed n -by- n matrix $\text{inv}(A)$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of A is zero, A is singular, and the inversion could not be completed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tptri` interface are as follows:

ap

Holds the array A of size $(n*(n+1)/2)$.

$uplo$

Must be '`U`' or '`L`'. The default value is '`U`'.

$diag$

Must be '`N`' or '`U`'. The default value is '`N`'.

Application Notes

The computed inverse X satisfies the following error bounds:

$$|XA - I| \leq c(n)\varepsilon |X| |A|$$

$$|X - A^{-1}| \leq c(n)\varepsilon |A^{-1}| |A| |X|,$$

where $c(n)$ is a modest linear function of n ; ε is the machine precision; I denotes the identity matrix.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors and $(4/3)n^3$ for complex flavors.

Routines for Matrix Equilibration

Routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

?geequ

Computes row and column scaling factors intended to equilibrate a general matrix and reduce its condition number.

Syntax**FORTRAN 77:**

```
call sggequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call dggequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call cggequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call zggequ( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
```

Fortran 95:

```
call geequ( a, r, c [,rowcnd] [,colcnd] [,amax] [,info] )
```

C:

```
lapack_int LAPACKE_sggequ( int matrix_layout, lapack_int m, lapack_int n, const float* a, lapack_int lda, float* r, float* c, float* rowcnd, float* colcnd, float* amax );
lapack_int LAPACKE_dggequ( int matrix_layout, lapack_int m, lapack_int n, const double* a, lapack_int lda, double* r, double* c, double* rowcnd, double* colcnd, double* amax );
lapack_int LAPACKE_cggequ( int matrix_layout, lapack_int m, lapack_int n,
const lapack_complex_float* a, lapack_int lda, float* r, float* c, float* rowcnd,
float* colcnd, float* amax );
lapack_int LAPACKE_zggequ( int matrix_layout, lapack_int m, lapack_int n,
const lapack_complex_double* a, lapack_int lda, double* r, double* c, double* rowcnd,
double* colcnd, double* amax );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes row and column scalings intended to equilibrate an m -by- n matrix A and reduce its condition number. The output array r returns the row scale factors and the array c the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r(i)*a_{ij}*c(j)$ have absolute value 1.

See [?laqge](#) auxiliary function that uses scaling factors computed by [?geequ](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m INTEGER. The number of rows of the matrix A ; $m \geq 0$.

n INTEGER. The number of columns of the matrix A ; $n \geq 0$.

a

REAL for sggequ
 DOUBLE PRECISION for dggequ
 COMPLEX for cggequ
 DOUBLE COMPLEX for zggequ.
Array: DIMENSION (*lda*, *).
 Contains the m -by- n matrix A whose equilibration factors are to be computed.
 The second dimension of *a* must be at least $\max(1, n)$.
lda
 INTEGER. The leading dimension of *a*; $lda \geq \max(1, m)$.

Output Parameters

r, c

REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
Arrays: $r(m), c(n)$.
 If $info = 0$, or $info > m$, the array *r* contains the row scale factors of the matrix A .
 If $info = 0$, the array *c* contains the column scale factors of the matrix A .

rowcnd

REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 If $info = 0$ or $info > m$, *rowcnd* contains the ratio of the smallest $r(i)$ to the largest $r(i)$.

colcnd

REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 If $info = 0$, *colcnd* contains the ratio of the smallest $c(i)$ to the largest $c(i)$.

amax

REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Absolute value of the largest element of the matrix A .

info

INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the *i*-th parameter had an illegal value.
 If $info = i$ and
 $i \leq m$, the *i*-th row of A is exactly zero;
 $i > m$, the $(i-m)$ th column of A is exactly zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geequ` interface are as follows:

- `a` Holds the matrix A of size (m, n) .
- `r` Holds the vector of length (m) .
- `c` Holds the vector of length n .

Application Notes

All the components of r and c are restricted to be between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the `?lamch` routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

If $\text{rowcnd} \geq 0.1$ and amax is neither too large nor too small, it is not worth scaling by r .

If $\text{colcnd} \geq 0.1$, it is not worth scaling by c .

If amax is very close to overflow or very close to underflow, the matrix A should be scaled.

?geequb

Computes row and column scaling factors restricted to a power of radix to equilibrate a general matrix and reduce its condition number.

Syntax

FORTRAN 77:

```
call sgeequb( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call dgeequb( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call cgeequb( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
call zgeequb( m, n, a, lda, r, c, rowcnd, colcnd, amax, info )
```

C:

```
lapack_int LAPACKE_sgeequb( int matrix_layout, lapack_int m, lapack_int n, const float* a, lapack_int lda, float* r, float* c, float* rowcnd, float* colcnd, float* amax );
lapack_int LAPACKE_dgeequb( int matrix_layout, lapack_int m, lapack_int n,
                           const double* a, lapack_int lda, double* r, double* c, double* rowcnd, double* colcnd,
                           double* amax );
lapack_int LAPACKE_cgeequb( int matrix_layout, lapack_int m, lapack_int n,
                           const lapack_complex_float* a, lapack_int lda, float* r, float* c, float* rowcnd,
                           float* colcnd, float* amax );
lapack_int LAPACKE_zgeequb( int matrix_layout, lapack_int m, lapack_int n,
                           const lapack_complex_double* a, lapack_int lda, double* r, double* c, double* rowcnd,
                           double* colcnd, double* amax );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes row and column scalings intended to equilibrate an m -by- n general matrix A and reduce its condition number. The output array r returns the row scale factors and the array c - the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij} = r(i) * a_{ij} * c(j)$ have an absolute value of at most the radix.

$r(i)$ and $c(j)$ are restricted to be a power of the radix between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of a but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the [?lamch](#) routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

This routine differs from [?geequ](#) by restricting the scaling factors to a power of the radix. Except for over- and underflow, scaling by these factors introduces no additional rounding errors. However, the scaled entries' magnitudes are no longer equal to approximately 1 but lie between $\sqrt{\text{radix}}$ and $1/\sqrt{\text{radix}}$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m	INTEGER. The number of rows of the matrix A ; $m \geq 0$.
n	INTEGER. The number of columns of the matrix A ; $n \geq 0$.
a	REAL for sgeequb DOUBLE PRECISION for dgeequb COMPLEX for cgeequb DOUBLE COMPLEX for zgeequb. Array: DIMENSION ($lda, *$). Contains the m -by- n matrix A whose equilibration factors are to be computed. The second dimension of a must be at least $\max(1, n)$.
lda	INTEGER. The leading dimension of a ; $lda \geq \max(1, m)$.

Output Parameters

r, c	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: $r(m), c(n)$.
--------	---

If $info = 0$, or $info > m$, the array r contains the row scale factors for the matrix A .

If $info = 0$, the array c contains the column scale factors for the matrix A .

rowcnd

REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.

If $info = 0$ or $info > m$, $rowcnd$ contains the ratio of the smallest $r(i)$ to the largest $r(i)$. If $rowcnd \geq 0.1$, and $amax$ is neither too large nor too small, it is not worth scaling by r .

colcnd

REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.

If $info = 0$, $colcnd$ contains the ratio of the smallest $c(i)$ to the largest $c(i)$. If $colcnd \geq 0.1$, it is not worth scaling by c .

amax

REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.

Absolute value of the largest element of the matrix A . If $amax$ is very close to overflow or very close to underflow, the matrix should be scaled.

info

INTEGER.
If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.
If $info = i$ and
 $i \leq m$, the i -th row of A is exactly zero;
 $i > m$, the $(i-m)$ -th column of A is exactly zero.

?gbequ

Computes row and column scaling factors intended to equilibrate a banded matrix and reduce its condition number.

Syntax

FORTRAN 77:

```
call sgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call dgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call cgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call zgbequ( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
```

Fortran 95:

```
call gbequ( ab, r, c [,kl] [,rowcnd] [,colcnd] [,amax] [,info] )
```

C:

```

lapack_int LAPACKE_sgbequ( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const float* ab, lapack_int ldab, float* r, float* c, float*
rowcnd, float* colcnd, float* amax );

lapack_int LAPACKE_dgbequ( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const double* ab, lapack_int ldab, double* r, double* c, double*
rowcnd, double* colcnd, double* amax );

lapack_int LAPACKE_cgbequ( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const lapack_complex_float* ab, lapack_int ldab, float* r, float*
c, float* rowcnd, float* colcnd, float* amax );

lapack_int LAPACKE_zgbequ( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const lapack_complex_double* ab, lapack_int ldab, double* r,
double* c, double* rowcnd, double* colcnd, double* amax );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes row and column scalings intended to equilibrate an m -by- n band matrix A and reduce its condition number. The output array r returns the row scale factors and the array c the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij} = r(i) * a_{ij} * c(j)$ have absolute value 1.

See [?laqgb](#) auxiliary function that uses scaling factors computed by [?gbequ](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows of the matrix A ; $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix A ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
<i>ab</i>	REAL for sgbequ DOUBLE PRECISION for dgbequ COMPLEX for cgbequ DOUBLE COMPLEX for zgbequ. Array, DIMENSION (<i>ldab</i> , *). Contains the original band matrix A stored in rows from 1 to $kl + ku + 1$. The second dimension of <i>ab</i> must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of *ab*; $ldab \geq kl+ku+1$.

Output Parameters

<i>r, c</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: $r(m), c(n)$. If $info = 0$, or $info > m$, the array <i>r</i> contains the row scale factors of the matrix <i>A</i> . If $info = 0$, the array <i>c</i> contains the column scale factors of the matrix <i>A</i> .
<i>rowcnd</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If $info = 0$ or $info > m$, <i>rowcnd</i> contains the ratio of the smallest $r(i)$ to the largest $r(i)$.
<i>colcnd</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If $info = 0$, <i>colcnd</i> contains the ratio of the smallest $c(i)$ to the largest $c(i)$.
<i>amax</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix <i>A</i> .
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the <i>i</i> -th parameter had an illegal value. If $info = i$ and $i \leq m$, the <i>i</i> -th row of <i>A</i> is exactly zero; $i > m$, the $(i-m)$ th column of <i>A</i> is exactly zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbequ` interface are as follows:

<i>ab</i>	Holds the array <i>A</i> of size $(kl+ku+1, n)$.
<i>r</i>	Holds the vector of length (m) .
<i>c</i>	Holds the vector of length n .
<i>kl</i>	If omitted, assumed $kl = ku$.

*ku*Restored as $ku = lda - kl - 1$.

Application Notes

All the components of r and c are restricted to be between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of A but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the [?lamch](#) routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

If $rowcnd \geq 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by r .

If $colcnd \geq 0.1$, it is not worth scaling by c .

If $amax$ is very close to overflow or very close to underflow, the matrix A should be scaled.

?gbequb

Computes row and column scaling factors restricted to a power of radix to equilibrate a banded matrix and reduce its condition number.

Syntax

FORTRAN 77:

```
call sgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call dgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call cgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
call zgbequb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, info )
```

C:

```
lapack_int LAPACKE_sgbequb( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const float* ab, lapack_int ldab, float* r, float* c, float*
rowcnd, float* colcnd, float* amax );

lapack_int LAPACKE_dgbequb( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const double* ab, lapack_int ldab, double* r, double* c, double*
rowcnd, double* colcnd, double* amax );

lapack_int LAPACKE_cgbequb( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const lapack_complex_float* ab, lapack_int ldab, float* r, float*
c, float* rowcnd, float* colcnd, float* amax );

lapack_int LAPACKE_zgbequb( int matrix_layout, lapack_int m, lapack_int n, lapack_int
kl, lapack_int ku, const lapack_complex_double* ab, lapack_int ldab, double* r,
double* c, double* rowcnd, double* colcnd, double* amax );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes row and column scalings intended to equilibrate an m -by- n banded matrix A and reduce its condition number. The output array r returns the row scale factors and the array c - the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij} = r(i) * a_{ij} * c(j)$ have an absolute value of at most the radix.

$r(i)$ and $c(j)$ are restricted to be a power of the radix between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of a but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the [?lamch](#) routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

This routine differs from [?gbequ](#) by restricting the scaling factors to a power of the radix. Except for over- and underflow, scaling by these factors introduces no additional rounding errors. However, the scaled entries' magnitudes are no longer equal to approximately 1 but lie between $\sqrt{\text{radix}}$ and $1/\sqrt{\text{radix}}$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m	INTEGER. The number of rows of the matrix A ; $m \geq 0$.
n	INTEGER. The number of columns of the matrix A ; $n \geq 0$.
k_1	INTEGER. The number of subdiagonals within the band of A ; $k_1 \geq 0$.
k_u	INTEGER. The number of superdiagonals within the band of A ; $k_u \geq 0$.
ab	REAL for sgbequb DOUBLE PRECISION for dgbequb COMPLEX for cgbequb DOUBLE COMPLEX for zgbequb. Array: DIMENSION ($ldab, *$). Contains the original banded matrix A stored in rows from 1 to $k_1 + k_u + 1$. The j -th column of A is stored in the j -th column of the array ab as follows: $ab(k_u+1+i-j, j) = a(i, j)$ for $\max(1, j-k_u) \leq i \leq \min(n, j+k_1)$. The second dimension of ab must be at least $\max(1, n)$.
$ldab$	INTEGER. The leading dimension of a ; $ldab \geq \max(1, m)$.

Output Parameters

r, c	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: $r(m), c(n)$.
--------	---

If $info = 0$, or $info > m$, the array r contains the row scale factors for the matrix A .

If $info = 0$, the array c contains the column scale factors for the matrix A .

rowcnd

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If $info = 0$ or $info > m$, *rowcnd* contains the ratio of the smallest $r(i)$ to the largest $r(i)$. If $rowcnd \geq 0.1$, and *amax* is neither too large nor too small, it is not worth scaling by r .

colcnd

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If $info = 0$, *colcnd* contains the ratio of the smallest $c(i)$ to the largest $c(i)$. If $colcnd \geq 0.1$, it is not worth scaling by c .

amax

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest element of the matrix A . If *amax* is very close to overflow or underflow, the matrix should be scaled.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the i -th diagonal element of A is nonpositive.

$i \leq m$, the i -th row of A is exactly zero;

$i > m$, the $(i-m)$ -th column of A is exactly zero.

?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

FORTRAN 77:

```
call spoequ( n, a, lda, s, scond, amax, info )
call dpoequ( n, a, lda, s, scond, amax, info )
call cpoequ( n, a, lda, s, scond, amax, info )
call zpoequ( n, a, lda, s, scond, amax, info )
```

Fortran 95:

```
call poequ( a, s [,scond] [,amax] [,info] )
```

C:

```
lapack_int LAPACKE_spoequ( int matrix_layout, lapack_int n, const float* a, lapack_int lda, float* s, float* scond, float* amax );
```

```

lapack_int LAPACKE_dpoequ( int matrix_layout, lapack_int n, const double* a, lapack_int
                          lda, double* s, double* scond, double* amax );

lapack_int LAPACKE_cpoequ( int matrix_layout, lapack_int n, const lapack_complex_float*
                           a, lapack_int lda, float* s, float* scond, float* amax );

lapack_int LAPACKE_zpoequ( int matrix_layout, lapack_int n,
                           const lapack_complex_double* a, lapack_int lda, double* s, double* scond, double*
                           amax );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix A and reduce its condition number (with respect to the two-norm). The output array s returns scale factors computed as

$$s_i = \frac{1}{\sqrt{A_{i,i}}}$$

These factors are chosen so that the scaled matrix B with elements $b_{ij} = s(i) * a_{ij} * s(j)$ has diagonal elements equal to 1.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

See [?laqsy](#) auxiliary function that uses scaling factors computed by [?poequ](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i>	REAL for spoequ DOUBLE PRECISION for dpoequ COMPLEX for cpoequ DOUBLE COMPLEX for zpoequ . Array: DIMENSION (<i>lda</i> , *).
	Contains the n -by- n symmetric or Hermitian positive definite matrix A whose scaling factors are to be computed. Only the diagonal elements of A are referenced.
	The second dimension of <i>a</i> must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of *a*; $lda \geq \max(1, n)$.

Output Parameters

<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (<i>n</i>). If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i> .
<i>scond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> (<i>i</i>) to the largest <i>s</i> (<i>i</i>).
<i>amax</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix <i>A</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of <i>A</i> is nonpositive.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `poequ` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>s</i>	Holds the vector of length <i>n</i> .

Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?poequb

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

FORTRAN 77:

```
call spoequb( n, a, lda, s, scond, amax, info )
call dpoequb( n, a, lda, s, scond, amax, info )
```

```
call cpoequb( n, a, lda, s, scond, amax, info )
call zpoequb( n, a, lda, s, scond, amax, info )
```

C:

```
lapack_int LAPACKE_spoequb( int matrix_layout, lapack_int n, const float* a, lapack_int
lda, float* s, float* scond, float* amax );
lapack_int LAPACKE_dpoequb( int matrix_layout, lapack_int n, const double* a,
lapack_int lda, double* s, double* scond, double* amax );
lapack_int LAPACKE_cpoequb( int matrix_layout, lapack_int n,
const lapack_complex_float* a, lapack_int lda, float* s, float* scond, float* amax );
lapack_int LAPACKE_zpoequb( int matrix_layout, lapack_int n,
const lapack_complex_double* a, lapack_int lda, double* s, double* scond, double*
amax );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix A and reduce its condition number (with respect to the two-norm).

These factors are chosen so that the scaled matrix B with elements $b(i,j)=s(i)*a(i,j)*s(j)$ has diagonal elements equal to 1. $s(i)$ is a power of two nearest to, but not exceeding $1/\sqrt{A(i,i)}$.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of the matrix A ; $n \geq 0$.
<i>a</i>	REAL for spoequb DOUBLE PRECISION for dpoequb COMPLEX for cpoequb DOUBLE COMPLEX for zpoequb. Array: DIMENSION (<i>lda</i> , *). Contains the n -by- n symmetric or Hermitian positive definite matrix A whose scaling factors are to be computed. Only the diagonal elements of A are referenced. The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, m)$.

Output Parameters

<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (<i>n</i>). If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i> .
<i>scond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> (<i>i</i>) to the largest <i>s</i> (<i>i</i>). If <i>scond</i> ≥ 0.1 , and <i>amax</i> is neither too large nor too small, it is not worth scaling by <i>s</i> .
<i>amax</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix <i>A</i> . If <i>amax</i> is very close to overflow or underflow, the matrix should be scaled.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of <i>A</i> is nonpositive.

?ppequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix in packed storage and reduce its condition number.

Syntax

FORTRAN 77:

```
call sppequ( uplo, n, ap, s, scond, amax, info )
call dppequ( uplo, n, ap, s, scond, amax, info )
call cppequ( uplo, n, ap, s, scond, amax, info )
call zppequ( uplo, n, ap, s, scond, amax, info )
```

Fortran 95:

```
call ppequ( ap, s [, scond] [, amax] [, uplo] [, info] )
```

C:

```
lapack_int LAPACKE_sppequ( int matrix_layout, char uplo, lapack_int n, const float* ap,
float* s, float* scond, float* amax);
lapack_int LAPACKE_dppequ( int matrix_layout, char uplo, lapack_int n, const double* ap,
double* s, double* scond, double* amax);
lapack_int LAPACKE_cppequ( int matrix_layout, char uplo, lapack_int n,
const lapack_complex_float* ap, float* s, float* scond, float* amax);
```

```
lapack_int LAPACKE_zppequ( int matrix_layout, char uplo, lapack_int n,
                           const lapack_complex_double* ap, double* s, double* scond, double* amax );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm). The output array s returns scale factors computed as

```
, ! : : : : :
```

These factors are chosen so that the scaled matrix B with elements $b_{ij} = s(i) * a_{ij} * s(j)$ has diagonal elements equal to 1.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

See [?laqsp](#) auxiliary function that uses scaling factors computed by [?ppequ](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is packed in the array <i>ap</i> :
	If <i>uplo</i> = 'U', the array <i>ap</i> stores the upper triangular part of the matrix A .
	If <i>uplo</i> = 'L', the array <i>ap</i> stores the lower triangular part of the matrix A .
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>ap</i>	REAL for sppequ DOUBLE PRECISION for dppequ COMPLEX for cppequ DOUBLE COMPLEX for zppequ. Array, DIMENSION at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in packed storage (see Matrix Storage Schemes).

Output Parameters

<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (<i>n</i>). If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i> .
<i>scond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> (<i>i</i>) to the largest <i>s</i> (<i>i</i>).
<i>amax</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Absolute value of the largest element of the matrix <i>A</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of <i>A</i> is nonpositive.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ppequ` interface are as follows:

<i>ap</i>	Holds the array <i>A</i> of size $(n * (n+1) / 2)$.
<i>s</i>	Holds the vector of length <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *s*.

If *amax* is very close to overflow or very close to underflow, the matrix *A* should be scaled.

?pbequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive-definite band matrix and reduce its condition number.

Syntax

FORTRAN 77:

```
call spbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
call dpbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
call cpbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
```

```
call zpbequ( uplo, n, kd, ab, ldab, s, scond, amax, info )
```

Fortran 95:

```
call pbequ( ab, s [,scond] [,amax] [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_spbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const float* ab, lapack_int ldab, float* s, float* scond, float* amax );
lapack_int LAPACKE_dpbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const double* ab, lapack_int ldab, double* s, double* scond, double* amax );
lapack_int LAPACKE_cpbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const lapack_complex_float* ab, lapack_int ldab, float* s, float* scond, float* amax );
lapack_int LAPACKE_zpbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
const lapack_complex_double* ab, lapack_int ldab, double* s, double* scond, double* amax );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix A in packed storage and reduce its condition number (with respect to the two-norm). The output array s returns scale factors computed as

$$s(i) = \frac{1}{\sqrt{a_{i,i}}}$$

These factors are chosen so that the scaled matrix B with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has diagonal elements equal to 1. This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

See [?laqsb](#) auxiliary function that uses scaling factors computed by [?pbequ](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is packed in the array *ab*:

If *uplo* = 'U', the array *ab* stores the upper triangular part of the matrix A .

If *uplo* = 'L', the array *ab* stores the lower triangular part of the matrix *A*.

n INTEGER. The order of matrix *A*; $n \geq 0$.

kd INTEGER. The number of superdiagonals or subdiagonals in the matrix *A*; $kd \geq 0$.

ab REAL for *spbequ*

DOUBLE PRECISION for *dpbequ*

COMPLEX for *cpbequ*

DOUBLE COMPLEX for *zpbequ*.

Array, DIMENSION (*ldab*, *).

The array *ap* contains either the upper or the lower triangular part of the matrix *A* (as specified by *uplo*) in *band storage* (see [Matrix Storage Schemes](#)).

The second dimension of *ab* must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of the array *ab*; $ldab \geq kd + 1$.

Output Parameters

s REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (*n*).

If *info* = 0, the array *s* contains the scale factors for *A*.

scond REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If *info* = 0, *scond* contains the ratio of the smallest *s*(*i*) to the largest *s*(*i*).

amax REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest element of the matrix *A*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, the *i*-th diagonal element of *A* is nonpositive.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *pbequ* interface are as follows:

<i>ab</i>	Holds the array A of size $(kd+1, n)$.
<i>s</i>	Holds the vector of length n .
<i>uplo</i>	Must be ' U ' or ' L '. The default value is ' U '.

Application Notes

If $scond \geq 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by s .

If $amax$ is very close to overflow or very close to underflow, the matrix A should be scaled.

?syequb

Computes row and column scaling factors intended to equilibrate a symmetric indefinite matrix and reduce its condition number.

Syntax

FORTRAN 77:

```
call ssyequb( uplo, n, a, lda, s, scond, amax, work, info )
call dsyequb( uplo, n, a, lda, s, scond, amax, work, info )
call csyequb( uplo, n, a, lda, s, scond, amax, work, info )
call zsyequb( uplo, n, a, lda, s, scond, amax, work, info )
```

C:

```
lapack_int LAPACKE_ssyequb( int matrix_layout, char uplo, lapack_int n, const float* a,
                             lapack_int lda, float* s, float* scond, float* amax );
lapack_int LAPACKE_dsyequb( int matrix_layout, char uplo, lapack_int n, const double* a,
                            lapack_int lda, double* s, double* scond, double* amax );
lapack_int LAPACKE_csyequb( int matrix_layout, char uplo, lapack_int n,
                           const lapack_complex_float* a, lapack_int lda, float* s, float* scond, float* amax );
lapack_int LAPACKE_zsyequb( int matrix_layout, char uplo, lapack_int n,
                           const lapack_complex_double* a, lapack_int lda, double* s, double* scond, double* amax );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric indefinite matrix A and reduce its condition number (with respect to the two-norm).

The array s contains the scale factors, $s(i) = 1/\sqrt{A(i,i)}$. These factors are chosen so that the scaled matrix B with elements $b(i,j)=s(i)*a(i,j)*s(j)$ has ones on the diagonal.

This choice of s puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored:

If $uplo = 'U'$, the array a stores the upper triangular part of the matrix A .

If $uplo = 'L'$, the array a stores the lower triangular part of the matrix A .

n

INTEGER. The order of the matrix A ; $n \geq 0$.

a, work

REAL for ssyequb

DOUBLE PRECISION for dsyequb

COMPLEX for csyequb

DOUBLE COMPLEX for zsyequb.

Array a : DIMENSION ($lda, *$).

Contains the n -by- n symmetric indefinite matrix A whose scaling factors are to be computed. Only the diagonal elements of A are referenced.

The second dimension of a must be at least $\max(1, n)$.

$work(*)$ is a workspace array. The dimension of $work$ is at least $\max(1, 3*n)$.

lda

INTEGER. The leading dimension of a ; $lda \geq \max(1, m)$.

Output Parameters

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (n).

If $info = 0$, the array s contains the scale factors for A .

scond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If $info = 0$, $scond$ contains the ratio of the smallest $s(i)$ to the largest $s(i)$. If $scond \geq 0.1$, and $amax$ is neither too large nor too small, it is not worth scaling by s .

amax

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest element of the matrix A . If $amax$ is very close to overflow or underflow, the matrix should be scaled.

<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of <i>A</i> is nonpositive.
-------------	---

?heequb

Computes row and column scaling factors intended to equilibrate a Hermitian indefinite matrix and reduce its condition number.

Syntax**FORTRAN 77:**

```
call cheequb( uplo, n, a, lda, s, scond, amax, work, info )
call zheequb( uplo, n, a, lda, s, scond, amax, work, info )
```

C:

```
lapack_int LAPACKE_cheequb( int matrix_layout, char uplo, lapack_int n,
const lapack_complex_float* a, lapack_int lda, float* s, float* scond, float* amax );
lapack_int LAPACKE_zheequb( int matrix_layout, char uplo, lapack_int n,
const lapack_complex_double* a, lapack_int lda, double* s, double* scond, double* amax );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes row and column scalings intended to equilibrate a Hermitian indefinite matrix *A* and reduce its condition number (with respect to the two-norm).

The array *s* contains the scale factors, $s(i) = 1/\sqrt{A(i,i)}$. These factors are chosen so that the scaled matrix *B* with elements $b(i,j)=s(i)*a(i,j)*s(j)$ has ones on the diagonal.

This choice of *s* puts the condition number of *B* within a factor *n* of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of <i>A</i> is stored: If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular part of the matrix <i>A</i> . If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular part of the matrix <i>A</i> .
-------------	---

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ; $n \geq 0$.
<i>a, work</i>	COMPLEX for cheequb DOUBLE COMPLEX for zheequb. Array <i>a</i> : DIMENSION (<i>lda</i> , *). Contains the n -by- n symmetric indefinite matrix <i>A</i> whose scaling factors are to be computed. Only the diagonal elements of <i>A</i> are referenced. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> is at least $\max(1, 3*n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, m)$.

Output Parameters

<i>s</i>	REAL for cheequb DOUBLE PRECISION for zheequb. Array, DIMENSION (<i>n</i>). If <i>info</i> = 0, the array <i>s</i> contains the scale factors for <i>A</i> .
<i>scond</i>	REAL for cheequb DOUBLE PRECISION for zheequb. If <i>info</i> = 0, <i>scond</i> contains the ratio of the smallest <i>s</i> (<i>i</i>) to the largest <i>s</i> (<i>i</i>). If <i>scond</i> ≥ 0.1 , and <i>amax</i> is neither too large nor too small, it is not worth scaling by <i>s</i> .
<i>amax</i>	REAL for cheequb DOUBLE PRECISION for zheequb. Absolute value of the largest element of the matrix <i>A</i> . If <i>amax</i> is very close to overflow or underflow, the matrix should be scaled.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the <i>i</i> -th diagonal element of <i>A</i> is nonpositive.

Driver Routines

Table "Driver Routines for Solving Systems of Linear Equations" lists the LAPACK driver routines for solving systems of linear equations with real or complex matrices.

Driver Routines for Solving Systems of Linear Equations

Matrix type, storage scheme	Simple Driver	Expert Driver	Expert Driver using Extra-Precise Iterative Refinement
general	?gesv	?gesvx	?gesvxx
general band	?gbsv	?gbsvx	?gbsvxx
general tridiagonal	?gtsv	?gtsvx	
diagonally dominant tridiagonal	?dtsvb		
symmetric/Hermitian positive-definite	?posv	?posvx	?posvxx
symmetric/Hermitian positive-definite, storage	?ppsv	?ppsvx	
symmetric/Hermitian positive-definite, band	?pbsv	?pbsvx	
symmetric/Hermitian positive-definite, tridiagonal	?ptsv	?ptsvx	
symmetric/Hermitian indefinite	?sysv/?hesv ?sysv_rook/ ?hesv_rook	?sysvx/?hesvx	?sysvxx/?hesvxx
symmetric/Hermitian indefinite, packed storage	?spsv/?hpsv	?spsvx/?hpsvx	
complex symmetric	?sysv ?sysv_rook	?sysvx	
complex symmetric, packed storage	?spsv	?spsvx	

In this table ? stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex). In the description of ?gesv and ?posv routines, the ? sign stands for combined character codes ds and zc for the mixed precision subroutines.

?gesv

Computes the solution to the system of linear equations with a square matrix A and multiple right-hand sides.

Syntax**FORTRAN 77:**

```
call sgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call dgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call cgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call zgesv( n, nrhs, a, lda, ipiv, b, ldb, info )
call dsgesv( n, nrhs, a, lda, ipiv, b, ldb, x, ldx, work, swork, iter, info )
```

```
call zcgesv( n, nrhs, a, lda, ipiv, b, ldb, x, idx, work, swork, rwork, iter, info )
```

Fortran 95:

```
call gesv( a, b [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>gesv( int matrix_layout, lapack_int n, lapack_int nrhs,
<datatype>* a, lapack_int lda, lapack_int* ipiv, <datatype>* b, lapack_int ldb );
lapack_int LAPACKE_dsgesv( int matrix_layout, lapack_int n, lapack_int nrhs, double* a,
lapack_int lda, lapack_int* ipiv, double* b, lapack_int ldb, double* x, lapack_int
idx, lapack_int* iter );
lapack_int LAPACKE_zcgesv( int matrix_layout, lapack_int n, lapack_int nrhs,
lapack_complex_double* a, lapack_int lda, lapack_int* ipiv, lapack_complex_double* b,
lapack_int ldb, lapack_complex_double* x, lapack_int idx, lapack_int* iter );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the system of linear equations $A \cdot X = B$, where A is an n -by- n matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The LU decomposition with partial pivoting and row interchanges is used to factor A as $A = P \cdot L \cdot U$, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. The factored form of A is then used to solve the system of equations $A \cdot X = B$.

The dsgesv and zcgesv are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (dsgesv) or single complex precision (zcgesv) and use this factorization within an iterative refinement procedure to produce a solution with double precision (dsgesv) / double complex precision (zcgesv) normwise backward error quality (see below). If the approach fails, the method switches to a double precision or double complex precision factorization respectively and computes the solution.

The iterative refinement is not going to be a winning strategy if the ratio single precision performance over double precision performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to ilaenv in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if

```
iter > itermax
```

or for all the right-hand sides:

```
rnmr < sqrt(n)*xnrm*anrm*eps*bwdmax
```

where

- `iter` is the number of the current iteration in the iterative refinement process
- `rnmr` is the infinity-norm of the residual
- `xnrm` is the infinity-norm of the solution
- `anrm` is the infinity-operator-norm of the matrix A
- `eps` is the machine epsilon returned by `dlamch ('Epsilon')`.

The values `itermax` and `bwdmax` are fixed to 30 and 1.0d+00 respectively.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The number of linear equations, that is, the order of the matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, that is, the number of columns of the matrix <i>B</i> ; $nrhs \geq 0$.
<i>a, b</i>	REAL for <code>s gesv</code> DOUBLE PRECISION for <code>d gesv</code> and <code>d s gesv</code> COMPLEX for <code>c gesv</code> DOUBLE COMPLEX for <code>z gesv</code> and <code>z c gesv</code> . Arrays: <i>a</i> (<i>lda</i> ,*) , <i>b</i> (<i>ldb</i> ,*). The array <i>a</i> contains the <i>n</i> -by- <i>n</i> coefficient matrix <i>A</i> . The array <i>b</i> contains the <i>n</i> -by- <i>nrhs</i> matrix of right hand side matrix <i>B</i> . The second dimension of <i>a</i> must be at least <code>max(1, n)</code> , the second dimension of <i>b</i> at least <code>max(1, nrhs)</code> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> ; $ldb \geq \max(1, n)$.
<i>ldx</i>	INTEGER. The leading dimension of the array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>work</i>	DOUBLE PRECISION for <code>d s gesv</code> DOUBLE COMPLEX for <code>z c gesv</code> . Workspace array, DIMENSION at least <code>max(1, n*nrhs)</code> . This array is used to hold the residual vectors.
<i>swork</i>	REAL for <code>d gesv</code> COMPLEX for <code>z gesv</code> . Workspace array, DIMENSION at least <code>max(1, n*(n+nrhs))</code> . This array is used to use the single precision matrix and the right-hand sides or solutions in single precision.
<i>rwork</i>	DOUBLE PRECISION. Workspace array, DIMENSION at least <code>max(1, n)</code> .

Output Parameters

<i>a</i>	Overwritten by the factors <i>L</i> and <i>U</i> from the factorization of $A = P^* L^* U$; the unit diagonal elements of <i>L</i> are not stored. If iterative refinement has been successfully used (<i>info</i> = 0 and <i>iter</i> ≥ 0), then <i>A</i> is unchanged.
----------	---

If double precision factorization has been used (*info*= 0 and *iter* < 0), then the array *A* contains the factors *L* and *U* from the factorization $A = P \cdot L \cdot U$; the unit diagonal elements of *L* are not stored.

b Overwritten by the solution matrix *X* for `dgesv`, `sgesv`, `zgesv`, `zgesv`. Unchanged for `dgesv` and `zgesv`.

ipiv INTEGER.

Array, DIMENSION at least `max(1, n)`. The pivot indices that define the permutation matrix *P*; row *i* of the matrix was interchanged with row *ipiv(i)*. Corresponds to the single precision factorization (if *info*= 0 and *iter* ≥ 0) or the double precision factorization (if *info*= 0 and *iter* < 0).

x DOUBLE PRECISION for `dgesv`
DOUBLE COMPLEX for `zgesv`.

Array, DIMENSION (*lidx*, *nrhs*). If *info* = 0, contains the *n*-by-*nrhs* solution matrix *X*.

iter INTEGER.

If *iter* < 0: iterative refinement has failed, double precision factorization has been performed

- If *iter* = -1: the routine fell back to full precision for implementation- or machine-specific reason
- If *iter* = -2: narrowing the precision induced an overflow, the routine fell back to full precision
- If *iter* = -3: failure of `sgetrf` for `dgesv`, or `cgetrf` for `zgesv`
- If *iter* = -31: stop the iterative refinement after the 30th iteration.

If *iter* > 0: iterative refinement has been successfully used. Returns the number of iterations.

info INTEGER. If *info*=0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, *U*(*i*, *i*) (computed in double precision for mixed precision subroutines) is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gesv` interface are as follows:

a Holds the matrix *A* of size (*n*, *n*).

b Holds the matrix *B* of size (*n*, *nrhs*).

*ipiv*Holds the vector of length n .**NOTE**Fortran 95 Interface is so far not available for the mixed precision subroutines `dgesv/zgesv`.**See Also**[ilaenv](#)[?lamch](#)[?getrf](#)**?gesvx**

Computes the solution to the system of linear equations with a square matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax**FORTRAN 77:**

```
call sgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, ferr, berr, work, iwork, info )

call dgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, ferr, berr, work, iwork, info )

call cgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, ferr, berr, work, rwork, info )

call zgesvx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
            ldx, rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gesvx( a, b, x [,af] [,ipiv] [,fact] [,trans] [,equed] [,r] [,c] [,ferr] [,berr]
            [,rcond] [,rpvgrw] [,info] )
```

C:

```
lapack_int LAPACKE_sgesvx( int matrix_layout, char fact, char trans, lapack_int n,
                            lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, lapack_int*
                            ipiv, char* equed, float* r, float* c, float* b, lapack_int ldb, float* x, lapack_int
                            ldx, float* rcond, float* ferr, float* berr, float* rpivot );

lapack_int LAPACKE_dgesvx( int matrix_layout, char fact, char trans, lapack_int n,
                            lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, lapack_int*
                            ipiv, char* equed, double* r, double* c, double* b, lapack_int ldb, double* x,
                            lapack_int ldx, double* rcond, double* ferr, double* berr, double* rpivot );

lapack_int LAPACKE_cgesvx( int matrix_layout, char fact, char trans, lapack_int n,
                            lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
                            lapack_int ldaf, lapack_int* ipiv, char* equed, float* r, float* c,
                            lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
                            float* rcond, float* ferr, float* berr, float* rpivot );
```

```
lapack_int LAPACKE_zgesvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, double* r, double* c,
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* rcond, double* ferr, double* berr, double* rpivot );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is an n -by- n matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gesvx performs the following steps:

1. If $fact = 'E'$, real scaling factors r and c are computed to equilibrate the system:

```
trans = 'N': diag(r)*A*diag(c)*inv(diag(c))*X = diag(r)*B
trans = 'T': (diag(r)*A*diag(c))T*inv(diag(r))*X = diag(c)*B
trans = 'C': (diag(r)*A*diag(c))H*inv(diag(r))*X = diag(c)*B
```

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $diag(r)*A*diag(c)$ and B by $diag(r)*B$ (if $trans='N'$) or $diag(c)*B$ (if $trans = 'T'$ or $'C'$).

2. If $fact = 'N'$ or $'E'$, the *LU* decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as $A = P \cdot L \cdot U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.
3. If some $U_{j,i} = 0$, so that U is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $diag(c)$ (if $trans = 'N'$) or $diag(r)$ (if $trans = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

The data types are given for the Fortran interface, except for $rpivot$. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $\text{fact} = \text{'F'}$: on entry, af and ipiv contain the factored form of A . If equed is not ' N ', the matrix A has been equilibrated with scaling factors given by r and c .

a , af , and ipiv are not modified.

If $\text{fact} = \text{'N'}$, the matrix A will be copied to af and factored.

If $\text{fact} = \text{'E'}$, the matrix A will be equilibrated if necessary, then copied to af and factored.

trans

CHARACTER*1. Must be ' N ', ' T ', or ' C '.

Specifies the form of the system of equations:

If $\text{trans} = \text{'N'}$, the system has the form $A*X = B$ (No transpose).

If $\text{trans} = \text{'T'}$, the system has the form $A^T*X = B$ (Transpose).

If $\text{trans} = \text{'C'}$, the system has the form $A^H*X = B$ (Transpose for real flavors, conjugate transpose for complex flavors).

n

INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.

nrhs

INTEGER. The number of right hand sides; the number of columns of the matrices B and X ; $\text{nrhs} \geq 0$.

a , af , b , work

REAL for sgetrf

DOUBLE PRECISION for dgetrf

COMPLEX for cgetrf

DOUBLE COMPLEX for zgetrf .

Arrays: $a(\text{lda},*)$, $\text{af}(\text{ldaf},*)$, $b(\text{ldb},*)$, $\text{work}(*)$.

The array a contains the matrix A . If $\text{fact} = \text{'F'}$ and equed is not ' N ', then A must have been equilibrated by the scaling factors in r and/or c . The second dimension of a must be at least $\max(1, n)$.

The array af is an input argument if $\text{fact} = \text{'F'}$. It contains the factored form of the matrix A , that is, the factors L and U from the factorization $A = P*L*U$ as computed by ?getrf . If equed is not ' N ', then af is the factored form of the equilibrated matrix A . The second dimension of af must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, \text{nrhs})$.

$\text{work}(*)$ is a workspace array. The dimension of work must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

lda

INTEGER. The leading dimension of a ; $\text{lda} \geq \max(1, n)$.

ldaf

INTEGER. The leading dimension of af ; $\text{ldaf} \geq \max(1, n)$.

ldb

INTEGER. The leading dimension of b ; $\text{ldb} \geq \max(1, n)$.

ipiv

INTEGER.

`Array, DIMENSION at least $\max(1, n)$.` The array `ipiv` is an input argument if `fact = 'F'`. It contains the pivot indices from the factorization $A = P \cdot L \cdot U$ as computed by `?getrf`; row i of the matrix was interchanged with row `ipiv(i)`.

`equed`

CHARACTER*1. Must be '`N`', '`R`', '`C`', or '`B`'.

`equed` is an input argument if `fact = 'F'`. It specifies the form of equilibration that was done:

If `equed = 'N'`, no equilibration was done (always true if `fact = 'N'`).

If `equed = 'R'`, row equilibration was done, that is, A has been premultiplied by `diag(r)`.

If `equed = 'C'`, column equilibration was done, that is, A has been postmultiplied by `diag(c)`.

If `equed = 'B'`, both row and column equilibration was done, that is, A has been replaced by `diag(r) * A * diag(c)`.

`r, c`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: `r(n)`, `c(n)`. The array `r` contains the row scale factors for A , and the array `c` contains the column scale factors for A . These arrays are input arguments if `fact = 'F'` only; otherwise they are output arguments.

If `equed = 'R'` or '`B`', A is multiplied on the left by `diag(r)`; if `equed = 'N'` or '`C`', r is not accessed.

If `fact = 'F'` and `equed = 'R'` or '`B`', each element of `r` must be positive.

If `equed = 'C'` or '`B`', A is multiplied on the right by `diag(c)`; if `equed = 'N'` or '`R`', `c` is not accessed.

If `fact = 'F'` and `equed = 'C'` or '`B`', each element of `c` must be positive.

`lidx`

INTEGER. The leading dimension of the output array `x`; `lidx $\geq \max(1, n)$` .

`iwork`

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.

`rwork`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION at least $\max(1, 2*n)$; used in complex flavors only.

Output Parameters

`x`

REAL for `sgetsvx`

DOUBLE PRECISION for `dgesvx`

COMPLEX for cgesvx

DOUBLE COMPLEX for zgesvx.

Array, DIMENSION ($l_{\text{dx}}, *$).

If $\text{info} = 0$ or $\text{info} = n+1$, the array x contains the solution matrix X to the *original* system of equations. Note that A and B are modified on exit if $\text{equed} \neq 'N'$, and the solution to the *equilibrated* system is:

$\text{diag}(C)^{-1}X$, if $\text{trans} = 'N'$ and $\text{equed} = 'C'$ or $'B'$; $\text{diag}(R)^{-1}X$, if $\text{trans} = 'T'$ or $'C'$ and $\text{equed} = 'R'$ or $'B'$. The second dimension of x must be at least $\max(1, nrhs)$.

a

Array a is not modified on exit if $\text{fact} = 'F'$ or $'N'$, or if $\text{fact} = 'E'$ and $\text{equed} = 'N'$. If $\text{equed} \neq 'N'$, A is scaled on exit as follows:
 $\text{equed} = 'R'$: $A = \text{diag}(R) * A$
 $\text{equed} = 'C'$: $A = A * \text{diag}(c)$
 $\text{equed} = 'B'$: $A = \text{diag}(R) * A * \text{diag}(c)$.

af

If $\text{fact} = 'N'$ or $'E'$, then af is an output argument and on exit returns the factors L and U from the factorization $A = PLU$ of the original matrix A (if $\text{fact} = 'N'$) or of the equilibrated matrix A (if $\text{fact} = 'E'$). See the description of a for the form of the equilibrated matrix.

b

Overwritten by $\text{diag}(r)^{*}B$ if $\text{trans} = 'N'$ and $\text{equed} = 'R'$ or $'B'$;
overwritten by $\text{diag}(c)^{*}B$ if $\text{trans} = 'T'$ or $'C'$ and $\text{equed} = 'C'$ or $'B'$;
not changed if $\text{equed} = 'N'$.

r, c

These arrays are output arguments if $\text{fact} \neq 'F'$. See the description of r, c in *Input Arguments* section.

$rcond$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. This condition is indicated by a return code of $\text{info} > 0$.

$ferr$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x^{(j)}$ (the j -th column of the solution matrix X). If x_{true} is the true solution corresponding to $x^{(j)}$, $ferr(j)$ is an estimated upper bound for the magnitude of the largest element in $(x^{(j)} - x_{\text{true}})$ divided by the magnitude of the

largest element in $x(j)$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

ipiv

If $fact = 'N'$ or $'E'$, then *ipiv* is an output argument and on exit contains the pivot indices from the factorization $A = P \cdot L \cdot U$ of the original matrix A (if $fact = 'N'$) or of the equilibrated matrix A (if $fact = 'E'$).

equed

If $fact \neq 'F'$, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

work, rwork, rpivot

On exit, *work(1)* for real flavors, or *rwork(1)* for complex flavors (the Fortran interface) and *rpivot* (the C interface), contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$. The "max absolute element" norm is used. If *work(1)* for real flavors, or *rwork(1)* for complex flavors is much less than 1, then the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution x , condition estimator $rcond$, and forward error bound *ferr* could be unreliable. If factorization fails with $0 < info \leq n$, then *work(1)* for real flavors, or *rwork(1)* for complex flavors contains the reciprocal pivot growth factor for the leading *info* columns of A .

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the *i*-th parameter had an illegal value.

If $info = i$, and $i \leq n$, then $U(i, i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n+1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine gesvx interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>af</i>	Holds the matrix AF of size (n, n) .
<i>ipiv</i>	Holds the vector of length n .
<i>r</i>	Holds the vector of length n . Default value for each element is $r(i) = 1.0_{\text{WP}}$.
<i>c</i>	Holds the vector of length n . Default value for each element is $c(i) = 1.0_{\text{WP}}$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>equed</i>	Must be 'N', 'B', 'C', or 'R'. The default value is 'N'.
<i>rpvgrw</i>	Real value that contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$.

?gesvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a square matrix A and multiple right-hand sides

Syntax

FORTRAN 77:

```
call sgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
             ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params,
             work, iwork, info )

call dgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
             ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params,
             work, iwork, info )

call cgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
             ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params,
             work, rwork, info )

call zgesvxx( fact, trans, n, nrhs, a, lda, af, ldaf, ipiv, equed, r, c, b, ldb, x,
             ldx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params,
             work, rwork, info )
```

C:

```

lapack_int LAPACKE_sgesvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, lapack_int* ipiv,
char* equed, float* r, float* c, float* b, lapack_int ldb, float* x, lapack_int ldx,
float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds, float*
err_bnds_norm, float* err_bnds_comp, lapack_int nparams, const float* params );

lapack_int LAPACKE_dgesvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, lapack_int* ipiv,
char* equed, double* r, double* c, double* b, lapack_int ldb, double* x,
lapack_int ldx, double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds,
double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double* params );

lapack_int LAPACKE_cgesvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, float* r, float* c,
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm,
float* err_bnds_comp, lapack_int nparams, const float* params );

lapack_int LAPACKE_zgesvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, double* r, double* c,
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds, double*
err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double* params );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is an n -by- n matrix, the columns of the matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?gesvxx performs the following steps:

1. If *fact* = 'E', scaling factors *r* and *c* are computed to equilibrate the system:

```

trans = 'N': diag(r)*A*diag(c)*inv(diag(c))*X = diag(r)*B
trans = 'T': (diag(r)*A*diag(c))T*inv(diag(r))*X = diag(c)*B
trans = 'C': (diag(r)*A*diag(c))H*inv(diag(r))*X = diag(c)*B

```

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(r) * A * \text{diag}(c)$ and B by $\text{diag}(r) * B$ (if $\text{trans} = 'N'$) or $\text{diag}(c) * B$ (if $\text{trans} = 'T'$ or $'C'$).

2. If $\text{fact} = 'N'$ or $'E'$, the LU decomposition is used to factor the matrix A (after equilibration if $\text{fact} = 'E'$) as $A = P * L * U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $\text{info} = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the rcond parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless $\text{params}(\text{la_linrx_itref_i})$ is set to zero, the routine applies iterative refinement to improve the computed solution matrix and calculate error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(c)$ (if $\text{trans} = 'N'$) or $\text{diag}(r)$ (if $\text{trans} = 'T'$ or $'C'$) so that it solves the original system before equilibration.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>fact</i>	CHARACTER*1. Must be ' <i>F</i> ', ' <i>N</i> ', or ' <i>E</i> '.
	Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.
	If $\text{fact} = 'F'$, on entry, af and ipiv contain the factored form of A . If equed is not ' <i>N</i> ', the matrix A has been equilibrated with scaling factors given by r and c . Parameters a , af , and ipiv are not modified.
	If $\text{fact} = 'N'$, the matrix A will be copied to af and factored.
	If $\text{fact} = 'E'$, the matrix A will be equilibrated, if necessary, copied to af and factored.
<i>trans</i>	CHARACTER*1. Must be ' <i>N</i> ', ' <i>T</i> ', or ' <i>C</i> '.
	Specifies the form of the system of equations:
	If $\text{trans} = 'N'$, the system has the form $A * X = B$ (No transpose).
	If $\text{trans} = 'T'$, the system has the form $A^{**T} * X = B$ (Transpose).
	If $\text{trans} = 'C'$, the system has the form $A^{**H} * X = B$ (Conjugate Transpose = Transpose for real flavors, Conjugate Transpose for complex flavors).
<i>n</i>	INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.
<i>a</i> , <i>af</i> , <i>b</i> , <i>work</i>	REAL for sgesvxx DOUBLE PRECISION for dgesvxx COMPLEX for cgesvxx

DOUBLE COMPLEX for zgesvxx.

Arrays: $a(1da, *)$, $af(1daf, *)$, $b(1db, *)$, $work(*)$.

The array a contains the matrix A . If $fact = 'F'$ and $equed$ is not ' N ', then A must have been equilibrated by the scaling factors in r and/or c . The second dimension of a must be at least $\max(1, n)$.

The array af is an input argument if $fact = 'F'$. It contains the factored form of the matrix A , that is, the factors L and U from the factorization $A = P \cdot L \cdot U$ as computed by [?getrf](#). If $equed$ is not ' N ', then af is the factored form of the equilibrated matrix A . The second dimension of af must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 4 * n)$ for real flavors, and at least $\max(1, 2 * n)$ for complex flavors.

$1da$

INTEGER. The leading dimension of a ; $1da \geq \max(1, n)$.

$1daf$

INTEGER. The leading dimension of af ; $1daf \geq \max(1, n)$.

$ipiv$

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array $ipiv$ is an input argument if $fact = 'F'$. It contains the pivot indices from the factorization $A = P \cdot L \cdot U$ as computed by [?getrf](#); row i of the matrix was interchanged with row $ipiv(i)$.

$equed$

CHARACTER*1. Must be ' N ', ' R ', ' C ', or ' B '.

$equed$ is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:

If $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$).

If $equed = 'R'$, row equilibration was done, that is, A has been premultiplied by $diag(r)$.

If $equed = 'C'$, column equilibration was done, that is, A has been postmultiplied by $diag(c)$.

If $equed = 'B'$, both row and column equilibration was done, that is, A has been replaced by $diag(r) \cdot A \cdot diag(c)$.

r, c

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: $r(n)$, $c(n)$. The array r contains the row scale factors for A , and the array c contains the column scale factors for A . These arrays are input arguments if $fact = 'F'$ only; otherwise they are output arguments.

If $equed = 'R'$ or ' B ', A is multiplied on the left by $diag(r)$; if $equed = 'N'$ or ' C ', r is not accessed.

If $fact = 'F'$ and $equed = 'R'$ or ' B ', each element of r must be positive.

If $\text{equed} = \text{'C'}$ or 'B' , A is multiplied on the right by $\text{diag}(c)$; if $\text{equed} = \text{'N'}$ or 'R' , c is not accessed.

If $\text{fact} = \text{'F'}$ and $\text{equed} = \text{'C'}$ or 'B' , each element of c must be positive.

Each element of r or c should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

ldx

INTEGER. The leading dimension of the output array x ; $ldx \geq \max(1, n)$.

n_err_bnds

INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See err_bnds_norm and err_bnds_comp descriptions in *Output Arguments* section below.

$nparams$

INTEGER. Specifies the number of parameters set in params . If ≤ 0 , the params array is never referenced and default values are used.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION $nparams$. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to $nparams$ are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass $nparams = 0$, which prevents the source code from accessing the params argument.

$\text{params}(\text{la_linrx_itref_i} = 1)$: Whether to perform iterative refinement or not. Default: 1.0

=0.0	No refinement is performed and no error bounds are computed.
------	--

=1.0	Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support DOUBLE PRECISION.
------	--

(Other values are reserved for future use.)

$\text{params}(\text{la_linrx_ithresh_i} = 2)$: Maximum number of residual computations allowed for refinement.

Default	10
---------	----

Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in err_bnds_norm and err_bnds_comp may no longer be trustworthy.
------------	---

`params(la_linx_cwise_i = 3)` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.

rwork

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION at least $\max(1, 3*n)$; used in complex flavors only.

Output Parameters

x

REAL for `sgevxx`

DOUBLE PRECISION for `dgevxx`

COMPLEX for `cgevxx`

DOUBLE COMPLEX for `zgevxx`.

Array, DIMENSION (*lidx*, *).

If *info* = 0, the array *x* contains the solution *n*-by-*nrhs* matrix *X* to the original system of equations. Note that *A* and *B* are modified on exit if *equed* ≠ 'N', and the solution to the equilibrated system is:

`inv(diag(c))*X`, if *trans* = 'N' and *equed* = 'C' or 'B'; or
`inv(diag(r))*X`, if *trans* = 'T' or 'C' and *equed* = 'R' or 'B'. The second dimension of *x* must be at least $\max(1, nrhs)$.

a

Array *a* is not modified on exit if *fact* = 'F' or 'N', or if *fact* = 'E' and *equed* = 'N'.

If *equed* ≠ 'N', *A* is scaled on exit as follows:

equed = 'R': *A* = `diag(r)*A`

equed = 'C': *A* = *A**`diag(c)`

equed = 'B': *A* = `diag(r)*A*diag(c)`.

af

If *fact* = 'N' or 'E', then *af* is an output argument and on exit returns the factors *L* and *U* from the factorization *A* = *PLU* of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E'). See the description of *a* for the form of the equilibrated matrix.

b

Overwritten by `diag(r)*B` if *trans* = 'N' and *equed* = 'R' or 'B';
overwritten by *trans* = 'T' or 'C' and *equed* = 'C' or 'B';
not changed if *equed* = 'N'.

r, c

These arrays are output arguments if *fact* ≠ 'F'. Each element of these arrays is a power of the radix. See the description of *r, c* in *Input Arguments* section.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

rpvgrw

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Contains the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element* norm is used. If this is much less than 1, the stability of the LU factorization of the (equilibrated) matrix A could be poor. This also means that the solution X , estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < \text{info} \leq n$, this parameter contains the reciprocal pivot growth factor for the leading info columns of A . In *?gesvx*, this quantity is returned in *work(1)*.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

err_bnds_norm

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION $(nrhs, n_{\text{err_bnds}})$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the i -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i, :)* corresponds to the i -th right-hand side.

The second index in *err_bnds_norm(:, err)* contains the following three fields:

<i>err=1</i>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt(n) * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt(n) * \text{dlamch}(\epsilon)$ for double precision flavors.
--------------	--

<i>err=2</i>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is
--------------	--

greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z.

Let $z = s^*a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION ($\text{nrhs}, n_{\text{err_bnds}}$). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{\text{true}_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params}(3) = 0.0$), then `err_bnds_comp` is not accessed. If $n_{\text{err_bnds}} < 3$, then at most the first $(:, n_{\text{err_bnds}})$ entries are returned.

The first index in `err_bnds_comp(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:, err)` contains the following three fields:

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

<code>err=2</code>	"Guaranteed" error bbound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z . Let $z = s^* (a^* \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a^* \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.
<code>ipiv</code>	If $\text{fact} = 'N'$ or $'E'$, then <code>ipiv</code> is an output argument and on exit contains the pivot indices from the factorization $A = P^* L^* U$ of the original matrix A (if $\text{fact} = 'N'$) or of the equilibrated matrix A (if $\text{fact} = 'E'$).
<code>equed</code>	If $\text{fact} \neq 'F'$, then <code>equed</code> is an output argument. It specifies the form of equilibration that was done (see the description of <code>equed</code> in <i>Input Arguments</i> section).
<code>params</code>	If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified
<code>info</code>	INTEGER. If $\text{info} = 0$, the execution is successful. The solution to every right-hand side is guaranteed. If $\text{info} = -i$, the i -th parameter had an illegal value. If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned. If $\text{info} = n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $\text{params}(3) = 0.0$, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that $\text{err_bnds_norm}(j, 1) = 0.0$ or $\text{err_bnds_comp}(j, 1) = 0.0$). See the definition of $\text{err_bnds_norm}(:, 1)$ and $\text{err_bnds_comp}(:, 1)$. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp .

?gbsv

Computes the solution to the system of linear equations with a band matrix A and multiple right-hand sides.

Syntax**FORTRAN 77:**

```
call sgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call dgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call cgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
call zgbsv( n, kl, ku, nrhs, ab, ldab, ipiv, b, ldb, info )
```

Fortran 95:

```
call gbsv( ab, b [,kl] [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>gbsv( int matrix_layout, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, <datatype>* ab, lapack_int ldab, lapack_int* ipiv, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the real or complex system of linear equations $A*X = B$, where A is an n -by- n band matrix with kl subdiagonals and ku superdiagonals, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The LU decomposition with partial pivoting and row interchanges is used to factor A as $A = L*U$, where L is a product of permutation and unit lower triangular matrices with kl subdiagonals, and U is upper triangular with $kl+ku$ superdiagonals. The factored form of A is then used to solve the system of equations $A*X = B$.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of A . The number of rows in B ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides. The number of columns in B ; $nrhs \geq 0$.
<i>ab, b</i>	REAL for sgbsv DOUBLE PRECISION for dgbsv

COMPLEX for `cgbsv`

DOUBLE COMPLEX for `zgbsv`.

Arrays: $ab(1:ldab, :)$, $b(1:ldb, :)$.

The array ab contains the matrix A in band storage (see [Matrix Storage Schemes](#)). The second dimension of ab must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$ldab$ INTEGER. The leading dimension of the array ab . ($ldab \geq 2k_1 + k_u + 1$)

ldb INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

ab Overwritten by L and U . The diagonal and $k_1 + k_u$ superdiagonals of U are stored in the first $1 + k_1 + k_u$ rows of ab . The multipliers used to form L are stored in the next k_1 rows.

b Overwritten by the solution matrix X .

$ipiv$ INTEGER.

Array, DIMENSION at least $\max(1, n)$. The pivot indices: row i was interchanged with row $ipiv(i)$.

$info$ INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, $U(i, i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbsv` interface are as follows:

ab Holds the array A of size $(2*k_1+k_u+1, n)$.

b Holds the matrix B of size $(n, nrhs)$.

$ipiv$ Holds the vector of length n .

k_1 If omitted, assumed $k_1 = k_u$.

ku Restored as $ku = lda - 2*k_1 - 1$.

?gbsvx

Computes the solution to the real or complex system of linear equations with a band matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax**FORTRAN 77:**

```
call sgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, iwork, info )
call dgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, iwork, info )
call cgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
call zgbsvx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c, b,
ldb, x, ldx, rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gbsvx( ab, b, x [,kl] [,afb] [,ipiv] [,fact] [,trans] [,equed] [,r] [,c] [,ferr]
[,berr] [,rcond] [,rpvgrw] [,info] )
```

C:

```
lapack_int LAPACKE_sgbsvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, float* ab, lapack_int ldab, float* afb,
lapack_int ldafb, lapack_int* ipiv, char* equed, float* r, float* c, float* b,
lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr, float* berr,
float* rpivot );
lapack_int LAPACKE_dgbsvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, double* ab, lapack_int ldab, double*
afb, lapack_int ldafb, lapack_int* ipiv, char* equed, double* r, double* c, double* b,
lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr,
double* rpivot );
lapack_int LAPACKE_cgbsvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_float* ab, lapack_int
ldab, lapack_complex_float* afb, lapack_int ldafb, lapack_int* ipiv, char* equed,
float* r, float* c, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* rcond, float* ferr, float* berr, float* rpivot );
lapack_int LAPACKE_zgbsvx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_double* ab, lapack_int
ldab, lapack_complex_double* afb, lapack_int ldafb, lapack_int* ipiv, char* equed,
double* r, double* c, lapack_complex_double* b, lapack_int ldb, lapack_complex_double*
x, lapack_int ldx, double* rcond, double* ferr, double* berr, double* rpivot );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A^*X = B$, $A^T*X = B$, or $A^H*X = B$, where A is a band matrix of order n with k_l subdiagonals and k_u superdiagonals, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gbsvx performs the following steps:

1. If $\text{fact} = \text{'E'}$, real scaling factors r and c are computed to equilibrate the system:

```
trans = 'N': diag(r)*A*diag(c) *inv(diag(c))*X = diag(r)*B
trans = 'T': (diag(r)*A*diag(c))^T *inv(diag(r))*X = diag(c)*B
trans = 'C': (diag(r)*A*diag(c))^H *inv(diag(r))*X = diag(c)*B
```

Whether the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(r)*A*\text{diag}(c)$ and B by $\text{diag}(r)*B$ (if $\text{trans} = \text{'N'}$) or $\text{diag}(c)*B$ (if $\text{trans} = \text{'T'}$ or 'C').

2. If $\text{fact} = \text{'N'}$ or 'E' , the *LU* decomposition is used to factor the matrix A (after equilibration if $\text{fact} = \text{'E'}$) as $A = L*U$, where L is a product of permutation and unit lower triangular matrices with k_l subdiagonals, and U is upper triangular with k_l+k_u superdiagonals.
3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $\text{info} = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $\text{info} = n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(c)$ (if $\text{trans} = \text{'N'}$) or $\text{diag}(r)$ (if $\text{trans} = \text{'T'}$ or 'C') so that it solves the original system before equilibration.

Input Parameters

The data types are given for the Fortran interface, except for rpivot . A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

fact

CHARACTER*1. Must be 'F' , 'N' , or 'E' .

Specifies whether the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $\text{fact} = \text{'F'}$: on entry, afb and ipiv contain the factored form of A . If equed is not 'N' , the matrix A is equilibrated with scaling factors given by r and c .

ab , afb , and ipiv are not modified.

If $\text{fact} = \text{'N'}$, the matrix A will be copied to afb and factored.

If $\text{fact} = \text{'E'}$, the matrix A will be equilibrated if necessary, then copied to afb and factored.

trans

CHARACTER*1. Must be 'N' , 'T' , or 'C' .

Specifies the form of the system of equations:

If $\text{trans} = \text{'N'}$, the system has the form $A^*X = B$ (No transpose).

If $\text{trans} = \text{'T'}$, the system has the form $A^T * X = B$ (Transpose).

If $\text{trans} = \text{'C'}$, the system has the form $A^H * X = B$ (Transpose for real flavors, conjugate transpose for complex flavors).

n

INTEGER. The number of linear equations, the order of the matrix A ; $n \geq 0$.

kl

INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.

ku

INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.

$nrhs$

INTEGER. The number of right hand sides, the number of columns of the matrices B and X ; $nrhs \geq 0$.

$ab, afb, b, work$

REAL for sgbsvx

DOUBLE PRECISION for dgbsvx

COMPLEX for cgbsvx

DOUBLE COMPLEX for zgbsvx.

Arrays: $ab(1:dab, *)$, $afb(1:dafb, *)$, $b(1:db, *)$, $work(*)$.

The array ab contains the matrix A in band storage (see [Matrix Storage Schemes](#)). The second dimension of ab must be at least $\max(1, n)$. If $\text{fact} = \text{'F'}$ and equed is not ' N ', then A must have been equilibrated by the scaling factors in r and/or c .

The array afb is an input argument if $\text{fact} = \text{'F'}$. The second dimension of afb must be at least $\max(1, n)$. It contains the factored form of the matrix A , that is, the factors L and U from the factorization $A = L * U$ as computed by [?gbtrf](#). U is stored as an upper triangular band matrix with $kl + ku$ superdiagonals in the first $1 + kl + ku$ rows of afb . The multipliers used during the factorization are stored in the next kl rows. If equed is not ' N ', then afb is the factored form of the equilibrated matrix A .

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

$ldab$

INTEGER. The leading dimension of ab ; $ldab \geq kl+ku+1$.

$ldafb$

INTEGER. The leading dimension of afb ; $ldafb \geq 2*kl+ku+1$.

ldb

INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

$ipiv$

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array $ipiv$ is an input argument if $\text{fact} = \text{'F'}$. It contains the pivot indices from the factorization $A = L * U$ as computed by [?gbtrf](#); row i of the matrix was interchanged with row $ipiv(i)$.

equed

CHARACTER*1. Must be 'N', 'R', 'C', or 'B'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').If *equed* = 'R', row equilibration was done, that is, *A* has been premultiplied by *diag(r)*.If *equed* = 'C', column equilibration was done, that is, *A* has been postmultiplied by *diag(c)*.If *equed* = 'B', both row and column equilibration was done, that is, *A* has been replaced by *diag(r)*A*diag(c)*.*r, c*

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: *r(n), c(n)*.The array *r* contains the row scale factors for *A*, and the array *c* contains the column scale factors for *A*. These arrays are input arguments if *fact* = 'F' only; otherwise they are output arguments.If *equed* = 'R' or 'B', *A* is multiplied on the left by *diag(r)*; if *equed* = 'N' or 'C', *r* is not accessed.If *fact* = 'F' and *equed* = 'R' or 'B', each element of *r* must be positive.If *equed* = 'C' or 'B', *A* is multiplied on the right by *diag(c)*; if *equed* = 'N' or 'R', *c* is not accessed.If *fact* = 'F' and *equed* = 'C' or 'B', each element of *c* must be positive.*ldx*INTEGER. The leading dimension of the output array *x*; *ldx* $\geq \max(1, n)$.*iwork*INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.*rwork*

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION at least $\max(1, n)$; used in complex flavors only.

Output Parameters

x

REAL for sgbsvx

DOUBLE PRECISION for dgbsvx

COMPLEX for cgbsvx

DOUBLE COMPLEX for zgbsvx.

Array, DIMENSION (*ldx, **).

If $info = 0$ or $info = n+1$, the array x contains the solution matrix X to the *original* system of equations. Note that A and B are modified on exit if $equed \neq 'N'$, and the solution to the *equilibrated* system is: $\text{inv}(\text{diag}(c)) * X$, if $trans = 'N'$ and $equed = 'C'$ or $'B'$; $\text{inv}(\text{diag}(r)) * X$, if $trans = 'T'$ or $'C'$ and $equed = 'R'$ or $'B'$.

The second dimension of x must be at least $\max(1, nrhs)$.

ab

Array *ab* is not modified on exit if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$.

If $equed \neq 'N'$, A is scaled on exit as follows:

$equed = 'R'$: $A = \text{diag}(r) * A$
 $equed = 'C'$: $A = A * \text{diag}(c)$
 $equed = 'B'$: $A = \text{diag}(r) * A * \text{diag}(c)$.

afb

If $fact = 'N'$ or $'E'$, then *afb* is an output argument and on exit returns details of the *LU* factorization of the original matrix A (if $fact = 'N'$) or of the equilibrated matrix A (if $fact = 'E'$). See the description of *ab* for the form of the equilibrated matrix.

b

Overwritten by $\text{diag}(r) * b$ if $trans = 'N'$ and $equed = 'R'$ or $'B'$;
overwritten by $\text{diag}(c) * b$ if $trans = 'T'$ or $'C'$ and $equed = 'C'$ or $'B'$;
not changed if $equed = 'N'$.

r, c

These arrays are output arguments if $fact \neq 'F'$. See the description of *r, c* in *Input Arguments* section.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix A after equilibration (if done).

If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j -th column of the solution matrix X). If *xtrue* is the true solution corresponding to $x(j)$, $ferr(j)$ is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, \text{nrhs})$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

ipiv

If $\text{fact} = \text{'N'}$ or 'E' , then *ipiv* is an output argument and on exit contains the pivot indices from the factorization $A = L*U$ of the original matrix A (if $\text{fact} = \text{'N'}$) or of the equilibrated matrix A (if $\text{fact} = \text{'E'}$).

equed

If $\text{fact} \neq \text{'F'}$, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

work, rwork, rpivot

On exit, *work(1)* for real flavors, or *rwork(1)* for complex flavors (the Fortran interface) and *rpivot* (the C interface), contains the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$. The "max absolute element" norm is used. If *work(1)* for real flavors, or *rwork(1)* for complex flavors is much less than 1, then the stability of the *LU* factorization of the (equilibrated) matrix A could be poor. This also means that the solution x , condition estimator *rcond*, and forward error bound *ferr* could be unreliable. If factorization fails with $0 < \text{info} \leq n$, then *work(1)* for real flavors, or *rwork(1)* for complex flavors contains the reciprocal pivot growth factor for the leading *info* columns of A .

info

INTEGER. If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th parameter had an illegal value.

If $\text{info} = i$, and $i \leq n$, then $U(i, i)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned. If $\text{info} = i$, and $i = n+1$, then U is nonsingular, but *rcond* is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of *rcond* would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gbsvx* interface are as follows:

ab

Holds the array A of size (k_1+k_u+1, n) .

b

Holds the matrix B of size (n, nrhs) .

x

Holds the matrix X of size (n, nrhs) .

afb

Holds the array AF of size $(2*k_1+k_u+1, n)$.

ipiv

Holds the vector of length n .

<i>r</i>	Holds the vector of length <i>n</i> . Default value for each element is <i>r</i> (<i>i</i>) = 1.0_WP.
<i>c</i>	Holds the vector of length <i>n</i> . Default value for each element is <i>c</i> (<i>i</i>) = 1.0_WP.
<i>ferr</i>	Holds the vector of length (<i>nrhs</i>).
<i>berr</i>	Holds the vector of length (<i>nrhs</i>).
<i>trans</i>	Must be 'N', 'C', or 'T'. The default value is 'N'.
<i>equed</i>	Must be 'N', 'B', 'C', or 'R'. The default value is 'N'.
<i>fact</i>	Must be 'N', 'E', or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.
<i>rpvgrw</i>	Real value that contains the reciprocal pivot growth factor norm(<i>A</i>)/norm(<i>U</i>).
<i>kl</i>	If omitted, assumed <i>kl</i> = <i>ku</i> .
<i>ku</i>	Restored as <i>ku</i> = <i>lda</i> - <i>kl</i> -1.

?gbsvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a banded matrix *A* and multiple right-hand sides

Syntax

FORTRAN 77:

```
call sgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c,
b, ldb, x, idx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp,
nparams, params, work, iwork, info )

call dgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c,
b, ldb, x, idx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp,
nparams, params, work, iwork, info )

call cgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c,
b, ldb, x, idx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp,
nparams, params, work, rwork, info )

call zgbsvxx( fact, trans, n, kl, ku, nrhs, ab, ldab, afb, ldafb, ipiv, equed, r, c,
b, ldb, x, idx, rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp,
nparams, params, work, rwork, info )
```

C:

```
lapack_int LAPACKE_sgbsvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, float* ab, lapack_int ldab, float* afb,
lapack_int ldafb, lapack_int* ipiv, char* equed, float* r, float* c, float* b,
lapack_int ldb, float* x, lapack_int idx, float* rcond, float* rpvgrw, float* berr,
lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams,
const float* params );
```

```

lapack_int LAPACKE_dgbsvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, double* ab, lapack_int ldab, double*
afb, lapack_int ldafb, lapack_int* ipiv, char* equed, double* r, double* c, double* b,
lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* rpvgrw, double*
berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int
nparams, const double* params );

lapack_int LAPACKE_cgbsvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_float* ab, lapack_int
ldab, lapack_complex_float* afb, lapack_int ldafb, lapack_int* ipiv, char* equed,
float* r, float* c, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds,
float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, const float* params );

lapack_int LAPACKE_zgbsvxx( int matrix_layout, char fact, char trans, lapack_int n,
lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_double* ab, lapack_int
ldab, lapack_complex_double* afb, lapack_int ldafb, lapack_int* ipiv, char* equed,
double* r, double* c, lapack_complex_double* b, lapack_int ldb, lapack_complex_double*
x, lapack_int ldx, double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds,
double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double*
params );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is an n -by- n banded matrix, the columns of the matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($\mathcal{O}(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $\mathcal{O}(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?gbsvxx performs the following steps:

1. If *fact* = 'E', scaling factors *r* and *c* are computed to equilibrate the system:

```

trans = 'N': diag(r)*A*diag(c)*inv(diag(c))*X = diag(r)*B
trans = 'T': (diag(r)*A*diag(c))^T*inv(diag(r))*X = diag(c)*B
trans = 'C': (diag(r)*A*diag(c))^H*inv(diag(r))*X = diag(c)*B

```

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(r) \cdot A \cdot \text{diag}(c)$ and B by $\text{diag}(r) \cdot B$ (if *trans*='N') or $\text{diag}(c) \cdot B$ (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as $A = P \cdot L \cdot U$, where P is a permutation matrix, L is a unit lower triangular matrix, and U is upper triangular.

3. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the $rcond$ parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless $\text{params}(\text{la_linrx_itref_i})$ is set to zero, the routine applies iterative refinement to improve the computed solution matrix and calculate error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(c)$ (if $\text{trans} = \text{'N'}$) or $\text{diag}(r)$ (if $\text{trans} = \text{'T'}$ or 'C') so that it solves the original system before equilibration.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>fact</i>	CHARACTER*1. Must be ' <i>F</i> ', ' <i>N</i> ', or ' <i>E</i> '.
	Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.
	If <i>fact</i> = ' <i>F</i> ', on entry, <i>afb</i> and <i>ipiv</i> contain the factored form of A . If <i>equed</i> is not ' <i>N</i> ', the matrix A has been equilibrated with scaling factors given by <i>r</i> and <i>c</i> . Parameters <i>ab</i> , <i>afb</i> , and <i>ipiv</i> are not modified.
	If <i>fact</i> = ' <i>N</i> ', the matrix A will be copied to <i>afb</i> and factored.
	If <i>fact</i> = ' <i>E</i> ', the matrix A will be equilibrated, if necessary, copied to <i>afb</i> and factored.
<i>trans</i>	CHARACTER*1. Must be ' <i>N</i> ', ' <i>T</i> ', or ' <i>C</i> '.
	Specifies the form of the system of equations:
	If <i>trans</i> = ' <i>N</i> ', the system has the form $A*X = B$ (No transpose).
	If <i>trans</i> = ' <i>T</i> ', the system has the form $A**T*X = B$ (Transpose).
	If <i>trans</i> = ' <i>C</i> ', the system has the form $A**H*X = B$ (Conjugate Transpose = Transpose for real flavors, Conjugate Transpose for complex flavors).
<i>n</i>	INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.
<i>ab</i> , <i>afb</i> , <i>b</i> , <i>work</i>	REAL for sgbsvxx DOUBLE PRECISION for dgbsvxx COMPLEX for cgbsvxx DOUBLE COMPLEX for zgbsvxx. Arrays: <i>ab</i> (<i>ldab</i> , *), <i>afb</i> (<i>ldafb</i> , *), <i>b</i> (<i>ldb</i> , *), <i>work</i> (*).

The array ab contains the matrix A in band storage, in rows 1 to $k_l + k_u + 1$. The j -th column of A is stored in the j -th column of the array ab as follows:

$$ab(k_u + 1 + i - j, j) = A(i, j) \text{ for } \max(1, j - k_u) \leq i \leq \min(n, j + k_l).$$

If $fact = 'F'$ and $equed$ is not ' N ', then AB must have been equilibrated by the scaling factors in r and/or c . The second dimension of a must be at least $\max(1, n)$.

The array afb is an input argument if $fact = 'F'$. It contains the factored form of the banded matrix A , that is, the factors L and U from the factorization $A = P^*L^*U$ as computed by [?gbtrf](#). U is stored as an upper triangular banded matrix with $k_l + k_u$ superdiagonals in rows 1 to $k_l + k_u + 1$. The multipliers used during the factorization are stored in rows $k_l + k_u + 2$ to $2*k_l + k_u + 1$. If $equed$ is not ' N ', then afb is the factored form of the equilibrated matrix A .

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

$ldab$

INTEGER. The leading dimension of the array ab ; $ldab \geq k_l + k_u + 1$.

$ldafb$

INTEGER. The leading dimension of the array afb ; $ldafb \geq 2*k_l + k_u + 1$.

$ipiv$

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array $ipiv$ is an input argument if $fact = 'F'$. It contains the pivot indices from the factorization $A = P^*L^*U$ as computed by [?gbtrf](#); row i of the matrix was interchanged with row $ipiv(i)$.

$equed$

CHARACTER*1. Must be ' N ', ' R ', ' C ', or ' B '.

$equed$ is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:

If $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$).

If $equed = 'R'$, row equilibration was done, that is, A has been premultiplied by $diag(r)$.

If $equed = 'C'$, column equilibration was done, that is, A has been postmultiplied by $diag(c)$.

If $equed = 'B'$, both row and column equilibration was done, that is, A has been replaced by $diag(r)^*A^*diag(c)$.

r, c

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: $r(n)$, $c(n)$. The array r contains the row scale factors for A , and the array c contains the column scale factors for A . These arrays are input arguments if $fact = 'F'$ only; otherwise they are output arguments.

If $equed = 'R'$ or ' B ', A is multiplied on the left by $diag(r)$; if $equed = 'N'$ or ' C ', r is not accessed.

If $\text{fact} = \text{'F'}$ and $\text{equed} = \text{'R'}$ or 'B' , each element of r must be positive.

If $\text{equed} = \text{'C'}$ or 'B' , A is multiplied on the right by $\text{diag}(c)$; if $\text{equed} = \text{'N'}$ or 'R' , c is not accessed.

If $\text{fact} = \text{'F'}$ and $\text{equed} = \text{'C'}$ or 'B' , each element of c must be positive.

Each element of r or c should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

ldx

INTEGER. The leading dimension of the output array x ; $ldx \geq \max(1, n)$.

n_err_bnds

INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See err_bnds_norm and err_bnds_comp descriptions in *Output Arguments* section below.

$nparams$

INTEGER. Specifies the number of parameters set in params . If ≤ 0 , the params array is never referenced and default values are used.

$params$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION $nparams$. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to $nparams$ are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass $nparams = 0$, which prevents the source code from accessing the params argument.

$\text{params}(\text{la_linrx_itref_i} = 1)$: Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0 No refinement is performed and no error bounds are computed.

=1.0 Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

$\text{params}(\text{la_linrx_ithresh_i} = 2)$: Maximum number of residual computations allowed for refinement.

Default	10
---------	----

Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a
------------	--

technique other than Gaussian elimination, the `quarantees` in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

`params(la_linrx_cwise_i = 3)` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

`iwork` INTEGER. Workspace array, DIMENSION at least `max(1, n)`; used in real flavors only.

`rwork` REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Workspace array, size at least `max(1, 2*n)`; used in complex flavors only.

Output Parameters

`x` REAL for `sgbsvxx`
DOUBLE PRECISION for `dgbsvxx`
COMPLEX for `cgbsvxx`
DOUBLE COMPLEX for `zgbsvxx`.
Array, DIMENSION (`ldx, *`).
If `info = 0`, the array `x` contains the solution n -by- $nrhs$ matrix X to the original system of equations. Note that A and B are modified on exit if `equed ≠ 'N'`, and the solution to the equilibrated system is:

`inv(diag(c))*X`, if `trans = 'N'` and `equed = 'C'` or `'B'`; or
`inv(diag(r))*X`, if `trans = 'T'` or `'C'` and `equed = 'R'` or `'B'`. The second dimension of `x` must be at least `max(1, nrhs)`.

`ab` Array `ab` is not modified on exit if `fact = 'F'` or `'N'`, or if `fact = 'E'` and `equed = 'N'`.

If `equed ≠ 'N'`, A is scaled on exit as follows:

`equed = 'R'`: $A = diag(r)*A$
`equed = 'C'`: $A = A*diag(c)$
`equed = 'B'`: $A = diag(r)*A*diag(c)$.

`afb` If `fact = 'N'` or `'E'`, then `afb` is an output argument and on exit returns the factors L and U from the factorization $A = PLU$ of the original matrix A (if `fact = 'N'`) or of the equilibrated matrix A (if `fact = 'E'`).

`b` Overwritten by `diag(r)*B` if `trans = 'N'` and `equed = 'R'` or `'B'`;
overwritten by `trans = 'T'` or `'C'` and `equed = 'C'` or `'B'`;
not changed if `equed = 'N'`.

<i>r, c</i>	These arrays are output arguments if <i>fact</i> ≠ 'F'. Each element of these arrays is a power of the radix. See the description of <i>r, c</i> in <i>Input Arguments</i> section.
<i>rcond</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision, in particular, if <i>rcond</i> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.</p>
<i>rpvgrw</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Contains the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The <i>max absolute element</i> norm is used. If this is much less than 1, the stability of the LU factorization of the (equilibrated) matrix <i>A</i> could be poor. This also means that the solution <i>X</i>, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < \text{info} \leq n$, this parameter contains the reciprocal pivot growth factor for the leading <i>info</i> columns of <i>A</i>. In ?gbsvx, this quantity is returned in <i>work</i>(1).</p>
<i>berr</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, DIMENSION at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes $x(j)$ an exact solution.</p>
<i>err_bnds_norm</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, DIMENSION $(nrhs, n_{\text{err_bnds}})$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:</p> <p>Normwise relative error in the <i>i</i>-th solution vector</p> $\frac{\max_j X_{\text{true}_{ji}} - X_{ji} }{\max_j X_{ji} }$ <p>The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.</p> <p>The first index in <i>err_bnds_norm(i, :)</i> corresponds to the <i>i</i>-th right-hand side.</p> <p>The second index in <i>err_bnds_norm(:, err)</i> contains the following three fields:</p>

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
<code>err=2</code>	"Guaranteed" error bbound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z . Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.
<code>err_bnds_comp</code>	<p>REAL for single precision flavors DOUBLE PRECISION for double precision flavors.</p> <p>Array, DIMENSION ($nrhs, n_{\text{err_bnds}}$). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:</p> <p>Componentwise relative error in the i-th solution vector:</p> $\max_j \frac{ X_{true_{ji}} - X_{ji} }{ X_{ji} }$ <p>The array is indexed by the right-hand side i, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params}(3) = 0.0$), then <code>err_bnds_comp</code> is not accessed. If $n_{\text{err_bnds}} < 3$, then at most the first $(:, n_{\text{err_bnds}})$ entries are returned.</p> <p>The first index in <code>err_bnds_comp(i, :)</code> corresponds to the i-th right-hand side.</p> <p>The second index in <code>err_bnds_comp(:, err)</code> contains the following three fields:</p>

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.
<code>err=2</code>	"Guaranteed" error bbound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z . Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.
<code>ipiv</code>	If <code>fact = 'N'</code> or <code>'E'</code> , then <code>ipiv</code> is an output argument and on exit contains the pivot indices from the factorization $A = P * L * U$ of the original matrix A (if <code>fact = 'N'</code>) or of the equilibrated matrix A (if <code>fact = 'E'</code>).
<code>equed</code>	If <code>fact ≠ 'F'</code> , then <code>equed</code> is an output argument. It specifies the form of equilibration that was done (see the description of <code>equed</code> in <i>Input Arguments</i> section).
<code>params</code>	If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. The solution to every right-hand side is guaranteed. If <code>info = -i</code> , the i -th parameter had an illegal value. If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; <code>rcond = 0</code> is returned. If <code>info = n+j</code> : The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested <code>params(3) = 0.0</code> , then the j -th right-hand side is the first with a normwise error bound

that is not guaranteed (the smallest j such that $\text{err_bnds_norm}(j, 1) = 0.0$ or $\text{err_bnds_comp}(j, 1) = 0.0$). See the definition of $\text{err_bnds_norm}(:, 1)$ and $\text{err_bnds_comp}(:, 1)$. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp .

?gtsv

Computes the solution to the system of linear equations with a tridiagonal matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sgtsv( n, nrhs, dl, d, du, b, ldb, info )
call dgtsv( n, nrhs, dl, d, du, b, ldb, info )
call cgtsv( n, nrhs, dl, d, du, b, ldb, info )
call zgtsv( n, nrhs, dl, d, du, b, ldb, info )
```

Fortran 95:

```
call gtsv( dl, d, du, b [,info] )
```

C:

```
lapack_int LAPACKE_<?>gtsv( int matrix_layout, lapack_int n, lapack_int nrhs,
<datatype>* dl, <datatype>* d, <datatype>* du, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the system of linear equations $A^T X = B$, where A is an n -by- n tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions. The routine uses Gaussian elimination with partial pivoting.

Note that the equation $A^T X = B$ may be solved by interchanging the order of the arguments du and dl .

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

n INTEGER. The order of A , the number of rows in B ; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.

dl, d, du, b REAL for sgtsv

DOUBLE PRECISION for dgtsv

COMPLEX for cgtsv

DOUBLE COMPLEX for zgtsv.

Arrays: $dl(n - 1)$, $d(n)$, $du(n - 1)$, $b(ldb, *)$.

The array dl contains the $(n - 1)$ subdiagonal elements of A .

The array d contains the diagonal elements of A .

The array du contains the $(n - 1)$ superdiagonal elements of A .

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

ldb

INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

dl

Overwritten by the $(n-2)$ elements of the second superdiagonal of the upper triangular matrix U from the LU factorization of A . These elements are stored in $dl(1), \dots, dl(n-2)$.

d

Overwritten by the n diagonal elements of U .

du

Overwritten by the $(n-1)$ elements of the first superdiagonal of U .

b

Overwritten by the solution matrix X .

$info$

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, $U(i, i)$ is exactly zero, and the solution has not been computed. The factorization has not been completed unless $i = n$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gtsv` interface are as follows:

dl Holds the vector of length $(n-1)$.

d Holds the vector of length n .

du Holds the vector of length $(n-1)$.

b Holds the matrix B of size $(n, nrhs)$.

?gtsvx

Computes the solution to the real or complex system of linear equations with a tridiagonal matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call sgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
             ldx, rcond, ferr, berr, work, iwork, info )

call dgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
             ldx, rcond, ferr, berr, work, iwork, info )

call cgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
             ldx, rcond, ferr, berr, work, rwork, info )

call zgtsvx( fact, trans, n, nrhs, dl, d, du, dlf, df, duf, du2, ipiv, b, ldb, x,
             ldx, rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call gtsvx( dl, d, du, b, x [,dlf] [,df] [,duf] [,du2] [,ipiv] [,fact] [,trans] [,ferr]
            [,berr] [,rcond] [,info] )
```

C:

```
lapack_int LAPACKE_sgtsvx( int matrix_layout, char fact, char trans, lapack_int n,
                            lapack_int nrhs, const float* dl, const float* d, const float* du, float* dlf, float*
                            df, float* duf, float* du2, lapack_int* ipiv, const float* b, lapack_int ldb, float*
                            x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_dgtsvx( int matrix_layout, char fact, char trans, lapack_int n,
                           lapack_int nrhs, const double* dl, const double* d, const double* du, double* dlf,
                           double* df, double* duf, double* du2, lapack_int* ipiv, const double* b, lapack_int
                           ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

lapack_int LAPACKE_cgtsvx( int matrix_layout, char fact, char trans, lapack_int n,
                           lapack_int nrhs, const lapack_complex_float* dl, const lapack_complex_float* d,
                           const lapack_complex_float* du, lapack_complex_float* dlf, lapack_complex_float* df,
                           lapack_complex_float* duf, lapack_complex_float* du2, lapack_int* ipiv,
                           const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
                           float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zgtsvx( int matrix_layout, char fact, char trans, lapack_int n,
                           lapack_int nrhs, const lapack_complex_double* dl, const lapack_complex_double* d,
                           const lapack_complex_double* du, lapack_complex_double* dlf, lapack_complex_double* df,
                           lapack_complex_double* duf, lapack_complex_double* du2, lapack_int* ipiv,
                           const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int
                           ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $A^*X = B$, $A^T*X = B$, or $A^H*X = B$, where A is a tridiagonal matrix of order n , the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gtsvx performs the following steps:

1. If $\text{fact} = \text{'N'}$, the LU decomposition is used to factor the matrix A as $A = L \cdot U$, where L is a product of permutation and unit lower bidiagonal matrices and U is an upper triangular matrix with nonzeros in only the main diagonal and first two superdiagonals.
2. If some $U_{i,i} = 0$, so that U is exactly singular, then the routine returns with $\text{info} = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $\text{info} = n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>fact</i>	CHARACTER*1. Must be 'F' or 'N'.
	Specifies whether or not the factored form of the matrix A has been supplied on entry.
	If $\text{fact} = \text{'F'}$: on entry, $d1f, df, duf, du2$, and $ipiv$ contain the factored form of A ; arrays $dl, d, du, d1f, df, duf, du2$, and $ipiv$ will not be modified.
	If $\text{fact} = \text{'N'}$, the matrix A will be copied to $d1f, df$, and duf and factored.
<i>trans</i>	CHARACTER*1. Must be 'N', 'T', or 'C'.
	Specifies the form of the system of equations:
	If $\text{trans} = \text{'N'}$, the system has the form $A \cdot X = B$ (No transpose).
	If $\text{trans} = \text{'T'}$, the system has the form $A^T \cdot X = B$ (Transpose).
	If $\text{trans} = \text{'C'}$, the system has the form $A^H \cdot X = B$ (Conjugate transpose).
<i>n</i>	INTEGER. The number of linear equations, the order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right hand sides, the number of columns of the matrices B and X ; $nrhs \geq 0$.
<i>d1, d, du, d1f, df, duf, du2, b, x, work</i>	REAL for sgtsvx DOUBLE PRECISION for dgtsvx COMPLEX for cgtsvx DOUBLE COMPLEX for zgtsvx. Arrays: $d1$, DIMENSION ($n - 1$), contains the subdiagonal elements of A . d , DIMENSION (n), contains the diagonal elements of A . du , DIMENSION ($n - 1$), contains the superdiagonal elements of A .

dif, DIMENSION (*n* - 1). If *fact* = 'F', then *dif* is an input argument and on entry contains the (*n* - 1) multipliers that define the matrix *L* from the *LU* factorization of *A* as computed by [?gttrf](#).

df, DIMENSION (*n*). If *fact* = 'F', then *df* is an input argument and on entry contains the *n* diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

duf, DIMENSION (*n* - 1). If *fact* = 'F', then *duf* is an input argument and on entry contains the (*n* - 1) elements of the first superdiagonal of *U*.

du2, DIMENSION (*n* - 2). If *fact* = 'F', then *du2* is an input argument and on entry contains the (*n* - 2) elements of the second superdiagonal of *U*.

b(*ldb**) contains the right-hand side matrix *B*. The second dimension of *b* must be at least $\max(1, \ nrhs)$.

x(*idx**) contains the solution matrix *X*. The second dimension of *x* must be at least $\max(1, \ nrhs)$.

work(*) is a workspace array.

DIMENSION of *work* must be at least $\max(1, \ 3*n)$ for real flavors and $\max(1, \ 2*n)$ for complex flavors.

ldb

INTEGER. The leading dimension of *b*; *ldb* $\geq \max(1, \ n)$.

idx

INTEGER. The leading dimension of *x*; *idx* $\geq \max(1, \ n)$.

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, \ n)$. If *fact* = 'F', then *ipiv* is an input argument and on entry contains the pivot indices, as returned by [?gttrf](#).

iwork

INTEGER. Workspace array, DIMENSION (*n*). Used for real flavors only.

rwork

REAL for *cgtsvx*

DOUBLE PRECISION for *zgtsvx*.

Workspace array, DIMENSION (*n*). Used for complex flavors only.

Output Parameters

x

REAL for *sgtsvx*

DOUBLE PRECISION for *dgtsvx*

COMPLEX for *cgtsvx*

DOUBLE COMPLEX for *zgtsvx*.

Array, DIMENSION (*idx*, *).

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X*. The second dimension of *x* must be at least $\max(1, \ nrhs)$.

d1f

If *fact* = 'N', then *d1f* is an output argument and on exit contains the $(n-1)$ multipliers that define the matrix *L* from the *LU* factorization of *A*.

df

If *fact* = 'N', then *df* is an output argument and on exit contains the n diagonal elements of the upper triangular matrix *U* from the *LU* factorization of *A*.

duf

If *fact* = 'N', then *duf* is an output argument and on exit contains the $(n-1)$ elements of the first superdiagonal of *U*.

du2

If *fact* = 'N', then *du2* is an output argument and on exit contains the $(n-2)$ elements of the second superdiagonal of *U*.

ipiv

The array *ipiv* is an output argument if *fact* = 'N' and, on exit, contains the pivot indices from the factorization $A = L \cdot U$; row *i* of the matrix was interchanged with row *ipiv(i)*. The value of *ipiv(i)* will always be *i* or *i+1*; *ipiv(i)=i* indicates a row interchange was not required.

rcond

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix *A*. If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of *info*>0.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the *j*-th column of the solution matrix *X*). If *xtrue* is the true solution corresponding to $x(j)$, *ferr(j)* is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of *A* or *B* that makes $x(j)$ an exact solution.

info

INTEGER. If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and $i \leq n$, then $U(i, i)$ is exactly zero. The factorization has not been completed unless $i = n$, but the factor *U* is exactly singular, so the solution and error bounds could not be computed;

$rcond = 0$ is returned. If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gtsvx` interface are as follows:

dl	Holds the vector of length $(n-1)$.
d	Holds the vector of length n .
du	Holds the vector of length $(n-1)$.
b	Holds the matrix B of size $(n, nrhs)$.
x	Holds the matrix X of size $(n, nrhs)$.
$d1f$	Holds the vector of length $(n-1)$.
df	Holds the vector of length n .
duf	Holds the vector of length $(n-1)$.
$du2$	Holds the vector of length $(n-2)$.
$ipiv$	Holds the vector of length n .
$ferr$	Holds the vector of length $(nrhs)$.
$berr$	Holds the vector of length $(nrhs)$.
$fact$	Must be ' N ' or ' F '. The default value is ' N '. If $fact = 'F'$, then the arguments $d1f$, df , duf , $du2$, and $ipiv$ must be present; otherwise, an error is returned.
$trans$	Must be ' N ', ' C ', or ' T '. The default value is ' N '.

?dtsvb

Computes the solution to the system of linear equations with a diagonally dominant tridiagonal matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sdtsvb( n, nrhs, dl, d, du, b, ldb, info )
call ddtsvb( n, nrhs, dl, d, du, b, ldb, info )
call cdtsvb( n, nrhs, dl, d, du, b, ldb, info )
```

```
call zdtsvb( n, nrhs, dl, d, du, b, ldb, info )
```

Fortran 95:

```
call dtsvb( dl, d, du, b [, info])
```

C:

```
void sdtsvb (const MKL_INT * n, const MKL_INT * nrhs, float * dl, float * d, const
float * du, float * b, const MKL_INT * ldb, MKL_INT * info );
void ddtsvb (const MKL_INT * n, const MKL_INT * nrhs, double * dl, double * d, const
double * du, double * b, const MKL_INT * ldb, MKL_INT * info );
void cdtsvb (const MKL_INT * n, const MKL_INT * nrhs, MKL_Complex8 * dl, MKL_Complex8 *
d, const MKL_Complex8 * du, MKL_Complex8 * b, const MKL_INT * ldb, MKL_INT * info );
void zdtsvb (const MKL_INT * n, const MKL_INT * nrhs, MKL_Complex16 * dl, MKL_Complex16 *
d, const MKL_Complex16 * du, MKL_Complex16 * b, const MKL_INT * ldb, MKL_INT *
info );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The ?dtsvb routine solves a system of linear equations $A^*X = B$ for X , where A is an n -by- n diagonally dominant tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions. The routine uses the BABE (Burning At Both Ends) algorithm.

Note that the equation $A^T X = B$ may be solved by interchanging the order of the arguments du and dl .

Input Parameters

<i>n</i>	INTEGER. The order of A , the number of rows in B ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>dl, d, du, b</i>	REAL for sdtsvb DOUBLE PRECISION for ddtsvb COMPLEX for cdtsvb DOUBLE COMPLEX for zdtsvb. Arrays: $dl(n - 1)$, $d(n)$, $du(n - 1)$, $b(ldb, *)$. The array dl contains the $(n - 1)$ subdiagonal elements of A . The array d contains the diagonal elements of A . The array du contains the $(n - 1)$ superdiagonal elements of A . The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

<i>dl</i>	Overwritten by the $(n-1)$ elements of the subdiagonal of the lower triangular matrices L_1, L_2 from the factorization of A .
<i>d</i>	Overwritten by the n diagonal element reciprocals of U .
<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	<p>INTEGER. If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p> <p>If $info = i$, u_{ii} is exactly zero, and the solution has not been computed. The factorization has not been completed unless $i = n$.</p>

Application Notes

A diagonally dominant tridiagonal system is defined such that $|d_i| > |dl_{i-1}| + |du_i|$ for any i :

$1 < i < n$, and $|d_1| > |du_1|$, $|d_n| > |dl_{n-1}|$

The underlying BABE algorithm is designed for diagonally dominant systems. Such systems have no numerical stability issue unlike the canonical systems that use elimination with partial pivoting (see [?gtsv](#)). The diagonally dominant systems are much faster than the canonical systems.

NOTE

- The current implementation of BABE has a potential accuracy issue on very small or large data close to the underflow or overflow threshold respectively. Scale the matrix before applying the solver in the case of such input data.
 - Applying the [?dtsvb](#) factorization to non-diagonally dominant systems may lead to an accuracy loss, or false singularity detected due to no pivoting.
-

?posv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sposv( uplo, n, nrhs, a, lda, b, ldb, info )
call dposv( uplo, n, nrhs, a, lda, b, ldb, info )
call cposv( uplo, n, nrhs, a, lda, b, ldb, info )
call zposv( uplo, n, nrhs, a, lda, b, ldb, info )
call dsposv( uplo, n, nrhs, a, lda, b, ldb, x, ldx, work, swork, iter, info )
call zcposv( uplo, n, nrhs, a, lda, b, ldb, x, ldx, work, swork, rwork, iter, info )
```

Fortran 95:

```
call posv( a, b [,uplo] [,info] )
```

C:

```

lapack_int LAPACKE_<?>posv( int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, <datatype>* a, lapack_int lda, <datatype>* b, lapack_int ldb );
lapack_int LAPACKE_dposv( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
double* a, lapack_int lda, double* b, lapack_int ldb, double* x, lapack_int ldx,
lapack_int* iter );
lapack_int LAPACKE_zcposv( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, lapack_int* iter );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the real or complex system of linear equations $A*X = B$, where A is an n -by- n symmetric/Hermitian positive-definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T * U$ (real flavors) and $A = U^H * U$ (complex flavors), if $uplo = 'U'$

or $A = L * L^T$ (real flavors) and $A = L * L^H$ (complex flavors), if $uplo = 'L'$,

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $A*X = B$.

The `dposv` and `zcposv` are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (`dposv`) or single complex precision (`zcposv`) and use this factorization within an iterative refinement procedure to produce a solution with double precision (`dposv`) / double complex precision (`zcposv`) normwise backward error quality (see below). If the approach fails, the method switches to a double precision or double complex precision factorization respectively and computes the solution.

The iterative refinement is not going to be a winning strategy if the ratio single precision/COMPLEX performance over double precision/DOUBLE COMPLEX performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to `ilaenv` in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if

`iter > itermax`

or for all the right-hand sides:

`rnrmr < sqrt(n) * xnrm * anrm * eps * bwdmax,`

where

- `iter` is the number of the current iteration in the iterative refinement process
- `rnrmr` is the infinity-norm of the residual
- `xnrm` is the infinity-norm of the solution
- `anrm` is the infinity-operator-norm of the matrix A
- `eps` is the machine epsilon returned by `dlamch ('Epsilon')`.

The values `itermax` and `bwdmax` are fixed to 30 and 1.0d+00 respectively.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored:If $uplo = 'U'$, the upper triangle of A is stored.If $uplo = 'L'$, the lower triangle of A is stored.*n*INTEGER. The order of matrix A ; $n \geq 0$.*nrhs*INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.*a, b*

REAL for sposv

DOUBLE PRECISION for dposv and dsposv.

COMPLEX for cposv

DOUBLE COMPLEX for zposv and zcposv.

Arrays: $a(1da, *)$, $b(1db, *)$. The array a contains the upper or the lower triangular part of the matrix A (see *uplo*). The second dimension of a must be at least $\max(1, n)$.

Note that in the case of zcposv the imaginary parts of the diagonal elements need not be set and are assumed to be zero.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.*lda*INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.*ldb*INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.*ldx*INTEGER. The leading dimension of the array x ; $ldx \geq \max(1, n)$.*work*

DOUBLE PRECISION for dsposv

DOUBLE COMPLEX for zcposv.

Workspace array, DIMENSION ($n * nrhs$). This array is used to hold the residual vectors.*swork*

REAL for ds gesv

COMPLEX for zc gesv.

Workspace array, DIMENSION ($n * (n + nrhs)$). This array is used to use the single precision matrix and the right-hand sides or solutions in single precision.*rwork*DOUBLE PRECISION. Workspace array, DIMENSION (n).

Output Parameters

<i>a</i>	If <i>info</i> = 0, the upper or lower triangular part of <i>a</i> is overwritten by the Cholesky factor <i>U</i> or <i>L</i> , as specified by <i>uplo</i> . If iterative refinement has been successfully used (<i>info</i> = 0 and <i>iter</i> ≥ 0), then <i>A</i> is unchanged. If double precision factorization has been used (<i>info</i> = 0 and <i>iter</i> < 0), then the array <i>A</i> contains the factors <i>L</i> and <i>U</i> from the Cholesky factorization; the unit diagonal elements of <i>L</i> are not stored.
<i>b</i>	Overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	INTEGER. Array, DIMENSION at least <code>max(1, n)</code> . The pivot indices that define the permutation matrix <i>P</i> ; row <i>i</i> of the matrix was interchanged with row <i>ipiv(i)</i> . Corresponds to the single precision factorization (if <i>info</i> = 0 and <i>iter</i> ≥ 0) or the double precision factorization (if <i>info</i> = 0 and <i>iter</i> < 0).
<i>x</i>	DOUBLE PRECISION for <code>dsposv</code> DOUBLE COMPLEX for <code>zcpovs</code> . Array, DIMENSION (<i>Idx</i> , <i>nrhs</i>). If <i>info</i> = 0, contains the <i>n</i> -by- <i>nrhs</i> solution matrix <i>X</i> .
<i>iter</i>	INTEGER. If <i>iter</i> < 0: iterative refinement has failed, double precision factorization has been performed <ul style="list-style-type: none"> • If <i>iter</i> = -1: the routine fell back to full precision for implementation- or machine-specific reason • If <i>iter</i> = -2: narrowing the precision induced an overflow, the routine fell back to full precision • If <i>iter</i> = -3: failure of <code>spotrf</code> for <code>dsposv</code>, or <code>cpotrf</code> for <code>zcpovs</code> • If <i>iter</i> = -31: stop the iterative refinement after the 30th iteration. If <i>iter</i> > 0: iterative refinement has been successfully used. Returns the number of iterations.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `posv` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

?posvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric or Hermitian positive-definite matrix *A*, and provides error bounds on the solution.

Syntax**FORTRAN 77:**

```
call sposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, iwork, info )
call dposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, iwork, info )
call cposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, rwork, info )
call zposvx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
ferr, berr, work, rwork, info )
```

Fortran 95:

```
call posvx( a, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

C:

```
lapack_int LAPACKE_sposvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, char* equed,
float* s, float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float*
ferr, float* berr );
lapack_int LAPACKE_dposvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, char* equed,
double* s, double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond,
double* ferr, double* berr );
lapack_int LAPACKE_cposvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, char* equed, float* s, lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );
lapack_int LAPACKE_zposvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, char* equed, double* s, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90

- C: mkl.h

Description

The routine uses the *Cholesky* factorization $A = U^T * U$ (real flavors) / $A = U^H * U$ (complex flavors) or $A = L * L^T$ (real flavors) / $A = L * L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A * X = B$, where A is a n -by- n real symmetric/Hermitian positive definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?posvx performs the following steps:

1. If $fact = 'E'$, real scaling factors s are computed to equilibrate the system:

$$\text{diag}(s) * A * \text{diag}(s) * \text{inv}(\text{diag}(s)) * X = \text{diag}(s) * B.$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$ and B by $\text{diag}(s) * B$.

2. If $fact = 'N'$ or $'E'$, the Cholesky decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as

$A = U^T * U$ (real), $A = U^H * U$ (complex), if $uplo = 'U'$,

or $A = L * L^T$ (real), $A = L * L^H$ (complex), if $uplo = 'L'$,

where U is an upper triangular matrix and L is a lower triangular matrix.

3. If the leading i -by- i principal minor is not positive-definite, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n + 1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

fact

CHARACTER*1. Must be '*F*', '*N*', or '*E*'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $fact = 'F'$: on entry, af contains the factored form of A . If $equed = 'Y'$, the matrix A has been equilibrated with scaling factors given by s .

a and af will not be modified.

If $fact = 'N'$, the matrix A will be copied to af and factored.

If $fact = 'E'$, the matrix A will be equilibrated if necessary, then copied to af and factored.

uplo

CHARACTER*1. Must be '*U*' or '*L*'.

Indicates whether the upper or lower triangular part of A is stored:

If $uplo = 'U'$, the upper triangle of A is stored.

If $uplo = 'L'$, the lower triangle of A is stored.

n

INTEGER. The order of matrix A ; $n \geq 0$.

$nrhs$

INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.

$a, af, b, work$

REAL for sposvx

DOUBLE PRECISION for dposvx

COMPLEX for cposvx

DOUBLE COMPLEX for zposvx.

Arrays: $a(1da, *), af(1daf, *), b(1db, *), work(*)$.

The array a contains the matrix A as specified by $uplo$. If $fact = 'F'$ and $equed = 'Y'$, then A must have been equilibrated by the scaling factors in s , and a must contain the equilibrated matrix $diag(s)^*A^*diag(s)$. The second dimension of a must be at least $\max(1, n)$.

The array af is an input argument if $fact = 'F'$. It contains the triangular factor U or L from the Cholesky factorization of A in the same storage format as A . If $equed$ is not ' N ', then af is the factored form of the equilibrated matrix $diag(s)^*A^*diag(s)$. The second dimension of af must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 3*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

$1da$

INTEGER. The leading dimension of a ; $1da \geq \max(1, n)$.

$1daf$

INTEGER. The leading dimension of af ; $1daf \geq \max(1, n)$.

$1db$

INTEGER. The leading dimension of b ; $1db \geq \max(1, n)$.

$equed$

CHARACTER*1. Must be ' N ' or ' Y '.

$equed$ is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:

if $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$);

if $equed = 'Y'$, equilibration was done, that is, A has been replaced by $diag(s)^*A^*diag(s)$.

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (n). The array s contains the scale factors for A . This array is an input argument if $\text{fact} = \text{'F'}$ only; otherwise it is an output argument.

If $\text{equed} = \text{'N'}$, s is not accessed.

If $\text{fact} = \text{'F'}$ and $\text{equed} = \text{'Y'}$, each element of s must be positive.

l_{dx}

INTEGER. The leading dimension of the output array x ; $l_{\text{dx}} \geq \max(1, n)$.

i_{work}

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.

r_{work}

REAL for cposvx

DOUBLE PRECISION for zposvx .

Workspace array, DIMENSION at least $\max(1, n)$; used in complex flavors only.

Output Parameters

x

REAL for sposvx

DOUBLE PRECISION for dposvx

COMPLEX for cposvx

DOUBLE COMPLEX for zposvx .

Array, DIMENSION ($l_{\text{dx}}, *$).

If $\text{info} = 0$ or $\text{info} = n+1$, the array x contains the solution matrix X to the original system of equations. Note that if $\text{equed} = \text{'Y'}$, A and B are modified on exit, and the solution to the equilibrated system is $\text{inv}(\text{diag}(s)) * X$. The second dimension of x must be at least $\max(1, nrhs)$.

a

Array a is not modified on exit if $\text{fact} = \text{'F'}$ or 'N' , or if $\text{fact} = \text{'E'}$ and $\text{equed} = \text{'N'}$.

If $\text{fact} = \text{'E'}$ and $\text{equed} = \text{'Y'}$, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$.

a_{f}

If $\text{fact} = \text{'N'}$ or 'E' , then a_{f} is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A = U^* * T * U$ or $A = L^* * L^{\prime *} * T$ (real routines), $A = U^* * H^* * U$ or $A = L^* * L^{\prime *} * H$ (complex routines) of the original matrix A (if $\text{fact} = \text{'N'}$), or of the equilibrated matrix A (if $\text{fact} = \text{'E'}$). See the description of a for the form of the equilibrated matrix.

b

Overwritten by $\text{diag}(s) * B$, if $\text{equed} = \text{'Y'}$; not changed if $\text{equed} = \text{'N'}$.

s

This array is an output argument if $\text{fact} \neq \text{'F'}$. See the description of s in Input Arguments section.

r_{cond}

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j -th column of the solution matrix X). If $xtrue$ is the true solution corresponding to $x(j)$, $ferr(j)$ is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

equed

If $fact \neq 'F'$, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `posvx` interface are as follows:

a

Holds the matrix A of size (n, n) .

<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>af</i>	Holds the matrix AF of size (n, n) .
<i>s</i>	Holds the vector of length n . Default value for each element is $s(i) = 1.0_{\text{WP}}$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be ' U ' or ' L '. The default value is ' U '.
<i>fact</i>	Must be ' N ', ' E ', or ' F '. The default value is ' N '. If <i>fact</i> = ' F ', then <i>af</i> must be present; otherwise, an error is returned.
<i>equed</i>	Must be ' N ' or ' Y '. The default value is ' N '.

?posvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a symmetric or Hermitian positive-definite matrix A applying the Cholesky factorization.

Syntax

FORTRAN 77:

```
call sposvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork,
info )
call dposvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, iwork,
info )
call cposvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork,
info )
call zposvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, equed, s, b, ldb, x, ldx, rcond,
rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work, rwork,
info )
```

C:

```
lapack_int LAPACKE_sposvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, char* equed,
float* s, float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float*
rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float*
err_bnds_comp, lapack_int nparams, const float* params );
```

```

lapack_int LAPACKE_dposvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, char* equed,
double* s, double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond,
double* rpvgrw, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp,
lapack_int nparams, const double* params );

lapack_int LAPACKE_cposvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, char* equed, float* s, lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* rcond, float* rpvgrw, float* berr,
lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams,
const float* params );

lapack_int LAPACKE_zposvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, char* equed, double* s, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* rpvgrw, double* berr,
lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams,
const double* params );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the *Cholesky* factorization $A = U^T * U$ (real flavors) / $A = U^H * U$ (complex flavors) or $A = L * L^T$ (real flavors) / $A = L * L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A * X = B$, where A is an n -by- n real symmetric/Hermitian positive definite matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($\mathcal{O}(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $\mathcal{O}(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?posvxx performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

$$\text{diag}(s) * A * \text{diag}(s) * \text{inv}(\text{diag}(s)) * X = \text{diag}(s) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$ and B by $\text{diag}(s) * B$.

2. If *fact* = 'N' or 'E', the Cholesky decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as

$$A = U^T * U \text{ (real)}, A = U^H * U \text{ (complex)}, \text{if } \text{uplo} = 'U',$$

$$\text{or } A = L * L^T \text{ (real)}, A = L * L^H \text{ (complex)}, \text{if } \text{uplo} = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix.

3. If the leading i -by- i principal minor is not positive-definite, the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the $rcond$ parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless $\text{params}(\text{la_linrx_itref_i})$ is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>fact</i>	CHARACTER*1. Must be 'F', 'N', or 'E'.
	Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.
	If <i>fact</i> = 'F', on entry, <i>af</i> contains the factored form of A . If <i>equed</i> is not 'N', the matrix A has been equilibrated with scaling factors given by <i>s</i> . Parameters <i>a</i> and <i>af</i> are not modified.
	If <i>fact</i> = 'N', the matrix A will be copied to <i>af</i> and factored.
	If <i>fact</i> = 'E', the matrix A will be equilibrated, if necessary, copied to <i>af</i> and factored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored:
	If <i>uplo</i> = 'U', the upper triangle of A is stored.
	If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.
<i>a</i> , <i>af</i> , <i>b</i> , <i>work</i>	REAL for sposvxx DOUBLE PRECISION for dposvxx COMPLEX for cposvxx DOUBLE COMPLEX for zposvxx. Arrays: <i>a</i> (<i>lda</i> , *), <i>af</i> (<i>ldaf</i> , *), <i>b</i> (<i>ldb</i> , *), <i>work</i> (*).
	The array <i>a</i> contains the matrix A as specified by <i>uplo</i> . If <i>fact</i> = 'F' and <i>equed</i> = 'Y', then A must have been equilibrated by the scaling factors in <i>s</i> , and <i>a</i> must contain the equilibrated matrix $\text{diag}(s) * A * \text{diag}(s)$. The second dimension of <i>a</i> must be at least $\max(1, n)$.

The array af is an input argument if $fact = 'F'$. It contains the triangular factor U or L from the Cholesky factorization of A in the same storage format as A . If $equed$ is not ' N ', then af is the factored form of the equilibrated matrix $diag(s)^*A^*diag(s)$. The second dimension of af must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

If $equed = 'N'$, s is not accessed.

If $fact = 'F'$ and $equed = 'Y'$, each element of s must be positive.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

lda

INTEGER. The leading dimension of the array a ; $lda \geq \max(1, n)$.

$ldaf$

INTEGER. The leading dimension of the array af ; $ldaf \geq \max(1, n)$.

$equed$

CHARACTER*1. Must be ' N ' or ' Y '.

$equed$ is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:

If $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$).

If $equed = 'Y'$, both row and column equilibration was done, that is, A has been replaced by $diag(s)^*A^*diag(s)$.

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (n). The array s contains the scale factors for A . This array is an input argument if $fact = 'F'$ only; otherwise it is an output argument.

If $equed = 'N'$, s is not accessed.

If $fact = 'F'$ and $equed = 'Y'$, each element of s must be positive.

Each element of s should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

idx

INTEGER. The leading dimension of the output array x ; $idx \geq \max(1, n)$.

n_err_bnds

INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See err_bnds_norm and err_bnds_comp descriptions in the *Output Arguments* section below.

$nparams$

INTEGER. Specifies the number of parameters set in $params$. If ≤ 0 , the $params$ array is never referenced and default values are used.

$params$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(la_linx_itref_i = 1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
------	--

=1.0	Use the extra-precise refinement algorithm.
------	---

(Other values are reserved for future use.)

params(la_linx_ithresh_i = 2) : Maximum number of residual computations allowed for refinement.

Default	10
---------	----

Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.
------------	---

params(la_linx_cwise_i = 3) : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

iwork INTEGER. Workspace array, DIMENSION at least max(1, n); used in real flavors only.

rwork REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION at least max(1, 3*n); used in complex flavors only.

Output Parameters

x REAL for sposvxx

DOUBLE PRECISION for dposvxx

COMPLEX for cposvxx

DOUBLE COMPLEX for zposvxx.

Array, DIMENSION (lidx,*).

If $info = 0$, the array x contains the solution n -by- $nrhs$ matrix X to the original system of equations. Note that A and B are modified on exit if $equed \neq 'N'$, and the solution to the equilibrated system is:

$\text{inv}(\text{diag}(s)) * X.$

a Array a is not modified on exit if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$.

If $fact = 'E'$ and $equed = 'Y'$, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$.

af If $fact = 'N'$ or $'E'$, then af is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A = U^{**} T^* U$ or $A = L^* L^{**} T$ (real routines), $A = U^{**} H^* U$ or $A = L^* L^{**} H$ (complex routines) of the original matrix A (if $fact = 'N'$), or of the equilibrated matrix A (if $fact = 'E'$). See the description of a for the form of the equilibrated matrix.

b If $equed = 'N'$, B is not modified.

If $equed = 'Y'$, B is overwritten by $\text{diag}(s) * B$.

s This array is an output argument if $fact \neq 'F'$. Each element of this array is a power of the radix. See the description of s in *Input Arguments* section.

$rcond$ REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

$rpvgrw$ REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Contains the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element* norm is used. If this is much less than 1, the stability of the *LU* factorization of the (equilibrated) matrix A could be poor. This also means that the solution X , estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < info \leq n$, this parameter contains the reciprocal pivot growth factor for the leading $info$ columns of A .

$berr$ REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

err_bnds_norm REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (*nrhs,n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

<i>err=1</i>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold <i>sqrt(n)*slamch(ε)</i> for single precision flavors and <i>sqrt(n)*dlamch(ε)</i> for double precision flavors.
<i>err=2</i>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold <i>sqrt(n)*slamch(ε)</i> for single precision flavors and <i>sqrt(n)*dlamch(ε)</i> for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<i>err=3</i>	Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold <i>sqrt(n)*slamch(ε)</i> for single precision flavors and <i>sqrt(n)*dlamch(ε)</i> for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are <i>1/(norm(1/z,inf)*norm(z,inf))</i> for some appropriately scaled matrix <i>Z</i> . Let <i>z=s*a</i> , where <i>s</i> scales each row by a power of the radix so all absolute row sums of <i>z</i> are approximately 1.
<i>err_bnds_comp</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (*nrhs,n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($params(3) = 0.0$), then err_bnds_comp is not accessed. If $n_err_bnds < 3$, then at most the first $(:, n_err_bnds)$ entries are returned.

The first index in $err_bnds_comp(i, :)$ corresponds to the i -th right-hand side.

The second index in $err_bnds_comp(:, err)$ contains the following three fields:

$err=1$ "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors.

$err=2$ "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

$err=3$ Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt(n) * slamch(\epsilon)$ for single precision flavors and $\sqrt(n) * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix Z .

Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

$equed$

If $fact \neq 'F'$, then $equed$ is an output argument. It specifies the form of equilibration that was done (see the description of $equed$ in *Input Arguments* section).

$params$

If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

info INTEGER. If *info* = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; *rcond* = 0 is returned.

If *info* = *n+j*: The solution corresponding to the *j*-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides *k* with *k* > *j* may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params(3)* = 0.0, then the *j*-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest *j* such that *err_bnds_norm(j,1)* = 0.0 or *err_bnds_comp(j,1)* = 0.0). See the definition of *err_bnds_norm(:,1)* and *err_bnds_comp(:,1)*. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

?ppsv

Computes the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed matrix *A* and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sppsv( uplo, n, nrhs, ap, b, ldb, info )
call dppsv( uplo, n, nrhs, ap, b, ldb, info )
call cppsv( uplo, n, nrhs, ap, b, ldb, info )
call zppsv( uplo, n, nrhs, ap, b, ldb, info )
```

Fortran 95:

```
call ppsv( ap, b [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>ppsv( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, <datatype>* ap, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for *X* the real or complex system of linear equations $A*X = B$, where *A* is an *n*-by-*n* real symmetric/Hermitian positive-definite matrix stored in packed format, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The Cholesky decomposition is used to factor *A* as

$A = U^T * U$ (real flavors) and $A = U^H * U$ (complex flavors), if *uplo* = 'U'

or $A = L^*L^T$ (real flavors) and $A = L^*L^H$ (complex flavors), if $uplo = 'L'$,

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

$uplo$	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
n	INTEGER. The order of matrix A ; $n \geq 0$.
$nrhs$	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
ap, b	REAL for sppsv DOUBLE PRECISION for dppsv COMPLEX for cppsv DOUBLE COMPLEX for zppsv. Arrays: $ap(*)$, $b(1:db, *)$. The array ap contains the upper or the lower triangular part of the matrix A (as specified by $uplo$) in <i>packed storage</i> (see Matrix Storage Schemes). The dimension of ap must be at least $\max(1, n(n+1)/2)$. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.
ldb	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

ap	If $info = 0$, the upper or lower triangular part of A in packed storage is overwritten by the Cholesky factor U or L , as specified by $uplo$.
b	Overwritten by the solution matrix X .
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ppsv` interface are as follows:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

?ppsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed matrix A , and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call sppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, idx, rcond, ferr,
berr, work, iwork, info )
call dppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, idx, rcond, ferr,
berr, work, iwork, info )
call cppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, idx, rcond, ferr,
berr, work, rwork, info )
call zppsvx( fact, uplo, n, nrhs, ap, afp, equed, s, b, ldb, x, idx, rcond, ferr,
berr, work, rwork, info )
```

Fortran 95:

```
call ppsvx( ap, b, x [,uplo] [,af] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

C:

```
lapack_int LAPACKE_sppsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, float* ap, float* afp, char* equed, float* s, float* b, lapack_int
ldb, float* x, lapack_int idx, float* rcond, float* ferr, float* berr );
lapack_int LAPACKE_dppsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, double* ap, double* afp, char* equed, double* s, double* b,
lapack_int ldb, double* x, lapack_int idx, double* rcond, double* ferr, double* berr );
lapack_int LAPACKE_cppsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float* ap, lapack_complex_float* afp, char* equed,
float* s, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int
idx, float* rcond, float* ferr, float* berr );
lapack_int LAPACKE_zppsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_double* ap, lapack_complex_double* afp, char* equed,
double* s, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x,
lapack_int idx, double* rcond, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the Cholesky factorization $A=U^T \cdot U$ (real flavors) / $A=U^H \cdot U$ (complex flavors) or $A=L \cdot L^T$ (real flavors) / $A=L \cdot L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is a n -by- n symmetric or Hermitian positive-definite matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ppsvx performs the following steps:

1. If $fact = 'E'$, real scaling factors s are computed to equilibrate the system:

```
diag(s)*A*diag(s)*inv(diag(s))*X = diag(s)*B.
```

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $diag(s)*A*diag(s)$ and B by $diag(s)*B$.

2. If $fact = 'N'$ or $'E'$, the Cholesky decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as

$A = U^T \cdot U$ (real), $A = U^H \cdot U$ (complex), if $uplo = 'U'$,

or $A = L \cdot L^T$ (real), $A = L \cdot L^H$ (complex), if $uplo = 'L'$,

where U is an upper triangular matrix and L is a lower triangular matrix.

3. If the leading i -by- i principal minor is not positive-definite, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.

4. The system of equations is solved for X using the factored form of A .

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix X is premultiplied by $diag(s)$ so that it solves the original system before equilibration.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

fact

CHARACTER*1. Must be ' F ', ' N ', or ' E '.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $fact = 'F'$: on entry, afp contains the factored form of A . If $equed = 'Y'$, the matrix A has been equilibrated with scaling factors given by s .
 ap and afp will not be modified.

If $fact = 'N'$, the matrix A will be copied to afp and factored.

If *fact* = 'E', the matrix *A* will be equilibrated if necessary, then copied to *afp* and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the upper triangle of *A* is stored.

If *uplo* = 'L', the lower triangle of *A* is stored.

n

INTEGER. The order of matrix *A*; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides; the number of columns in *B*; $nrhs \geq 0$.

ap, afp, b, work

REAL for *sppsvx*

DOUBLE PRECISION for *dppsvx*

COMPLEX for *cppsvx*

DOUBLE COMPLEX for *zppsvx*.

Arrays: *ap*(*), *afp*(*), *b*(*ldb*, *), *work*(*).

The array *ap* contains the upper or lower triangle of the original symmetric/Hermitian matrix *A* in *packed storage* (see [Matrix Storage Schemes](#)). In case when *fact* = 'F' and *equed* = 'Y', *ap* must contain the equilibrated matrix *diag(s)*A*diag(s)*.

The array *afp* is an input argument if *fact* = 'F' and contains the triangular factor *U* or *L* from the Cholesky factorization of *A* in the same storage format as *A*. If *equed* is not 'N', then *afp* is the factored form of the equilibrated matrix *A*.

The array *b* contains the matrix *B* whose columns are the right-hand sides for the systems of equations.

work(*) is a workspace array.

The dimension of arrays *ap* and *afp* must be at least $\max(1, n(n + 1)/2)$; the second dimension of *b* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldb

INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

equed

CHARACTER*1. Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

if *equed* = 'N', no equilibration was done (always true if *fact* = 'N');

if *equed* = 'Y', equilibration was done, that is, *A* has been replaced by *diag(s)*A*diag(s)*.

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (n). The array s contains the scale factors for A . This array is an input argument if $\text{fact} = \text{'F'}$ only; otherwise it is an output argument.

If $\text{equed} = \text{'N'}$, s is not accessed.

If $\text{fact} = \text{'F'}$ and $\text{equed} = \text{'Y'}$, each element of s must be positive.

ldx

INTEGER. The leading dimension of the output array x ; $\text{ldx} \geq \max(1, n)$.

$iwork$

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.

$rwork$

REAL for cppsvx ;

DOUBLE PRECISION for zppsvx .

Workspace array, DIMENSION at least $\max(1, n)$; used in complex flavors only.

Output Parameters

x

REAL for sppsvx

DOUBLE PRECISION for dppsvx

COMPLEX for cppsvx

DOUBLE COMPLEX for zppsvx .

Array, DIMENSION ($\text{ldx}, *$).

If $\text{info} = 0$ or $\text{info} = n+1$, the array x contains the solution matrix X to the original system of equations. Note that if $\text{equed} = \text{'Y'}$, A and B are modified on exit, and the solution to the equilibrated system is $\text{inv}(\text{diag}(s)) * X$. The second dimension of x must be at least $\max(1, \text{nrhs})$.

ap

Array ap is not modified on exit if $\text{fact} = \text{'F'}$ or 'N' , or if $\text{fact} = \text{'E'}$ and $\text{equed} = \text{'N'}$.

If $\text{fact} = \text{'E'}$ and $\text{equed} = \text{'Y'}$, A is overwritten by $\text{diag}(s) * A * \text{diag}(s)$.

afp

If $\text{fact} = \text{'N'}$ or 'E' , then afp is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ (real routines), $A = U^H * U$ or $A = L * L^H$ (complex routines) of the original matrix A (if $\text{fact} = \text{'N'}$), or of the equilibrated matrix A (if $\text{fact} = \text{'E'}$). See the description of ap for the form of the equilibrated matrix.

b

Overwritten by $\text{diag}(s) * B$, if $\text{equed} = \text{'Y'}$; not changed if $\text{equed} = \text{'N'}$.

s

This array is an output argument if $\text{fact} \neq \text{'F'}$. See the description of s in Input Arguments section.

$rcond$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j -th column of the solution matrix X). If $xtrue$ is the true solution corresponding to $x(j)$, $ferr(j)$ is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

equed

If $fact \neq 'F'$, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

info

INTEGER. If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ppsvx` interface are as follows:

ap

Holds the array A of size $(n^*(n+1)/2)$.

<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>x</i>	Holds the matrix X of size $(n, nrhs)$.
<i>afp</i>	Holds the matrix AF of size $(n^*(n+1)/2)$.
<i>s</i>	Holds the vector of length n . Default value for each element is $s(i) = 1.0_{\text{WP}}$.
<i>ferr</i>	Holds the vector of length $(nrhs)$.
<i>berr</i>	Holds the vector of length $(nrhs)$.
<i>uplo</i>	Must be ' U ' or ' L '. The default value is ' U '.
<i>fact</i>	Must be ' N ', ' E ', or ' F '. The default value is ' N '. If <i>fact</i> = ' F ', then <i>af</i> must be present; otherwise, an error is returned.
<i>equed</i>	Must be ' N ' or ' Y '. The default value is ' N '.

?pbsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite band matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call spbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call dpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call cpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
call zpbsv( uplo, n, kd, nrhs, ab, ldab, b, ldb, info )
```

Fortran 95:

```
call pbsv( ab, b [,uplo] [,info] )
```

C:

```
lapack_int LAPACKE_<?>pbsv( int matrix_layout, char uplo, lapack_int n, lapack_int kd,
lapack_int nrhs, <datatype>* ab, lapack_int ldab, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the real or complex system of linear equations $A*X = B$, where A is an n -by- n symmetric/Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The Cholesky decomposition is used to factor A as

$A = U^T * U$ (real flavors) and $A = U^H * U$ (complex flavors), if *uplo* = ' U '

or $A = L^*L^T$ (real flavors) and $A = L^*L^H$ (complex flavors), if $uplo = 'L'$,

where U is an upper triangular band matrix and L is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as A . The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored:
	If $uplo = 'U'$, the upper triangle of A is stored.
	If $uplo = 'L'$, the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>kd</i>	INTEGER. The number of superdiagonals of the matrix A if $uplo = 'U'$, or the number of subdiagonals if $uplo = 'L'$; $kd \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>ab, b</i>	REAL for spbsv DOUBLE PRECISION for dpbsv COMPLEX for cpbsv DOUBLE COMPLEX for zpbsv. Arrays: $ab(1dab, *)$, $b(1db, *)$. The array ab contains the upper or the lower triangular part of the matrix A (as specified by <i>uplo</i>) in <i>band storage</i> (see Matrix Storage Schemes). The second dimension of ab must be at least $\max(1, n)$. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.
<i>1dab</i>	INTEGER. The leading dimension of the array ab ; $1dab \geq kd + 1$.
<i>1db</i>	INTEGER. The leading dimension of b ; $1db \geq \max(1, n)$.

Output Parameters

<i>ab</i>	The upper or lower triangular part of A (in band storage) is overwritten by the Cholesky factor U or L , as specified by <i>uplo</i> , in the same storage format as A .
<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbsv` interface are as follows:

<code>ab</code>	Holds the array A of size $(kd+1, n)$.
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

?pbsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive-definite band matrix A , and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call spbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )
call dpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, iwork, info )
call cpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
call zpbsvx( fact, uplo, n, kd, nrhs, ab, ldab, afb, ldafb, equed, s, b, ldb, x, ldx,
rcond, ferr, berr, work, rwork, info )
```

Fortran 95:

```
call pbsvx( ab, b, x [,uplo] [,afb] [,fact] [,equed] [,s] [,ferr] [,berr] [,rcond]
[,info] )
```

C:

```
lapack_int LAPACKE_spbsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int kd, lapack_int nrhs, float* ab, lapack_int ldab, float* afb, lapack_int
ldafb, char* equed, float* s, float* b, lapack_int ldb, float* x, lapack_int ldx,
float* rcond, float* ferr, float* berr );
lapack_int LAPACKE_dpbsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int kd, lapack_int nrhs, double* ab, lapack_int ldab, double* afb, lapack_int
ldafb, char* equed, double* s, double* b, lapack_int ldb, double* x, lapack_int ldx,
double* rcond, double* ferr, double* berr );
```

```

lapack_int LAPACKE_cpbsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int kd, lapack_int nrhs, lapack_complex_float* ab, lapack_int ldab,
lapack_complex_float* afb, lapack_int ldafb, char* equed, float* s,
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int idx,
float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zpbsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int kd, lapack_int nrhs, lapack_complex_double* ab, lapack_int ldab,
lapack_complex_double* afb, lapack_int ldafb, char* equed, double* s,
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int idx,
double* rcond, double* ferr, double* berr );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the Cholesky factorization $A=U^T \cdot U$ (real flavors) / $A=U^H \cdot U$ (complex flavors) or $A=L \cdot L^T$ (real flavors) / $A=L \cdot L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is a n -by- n symmetric or Hermitian positive definite band matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?pbsvx performs the following steps:

1. If $fact = 'E'$, real scaling factors s are computed to equilibrate the system:

$$diag(s) \cdot A \cdot diag(s) \cdot \text{inv}(diag(s)) \cdot X = diag(s) \cdot B.$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $diag(s) \cdot A \cdot diag(s)$ and B by $diag(s) \cdot B$.

2. If $fact = 'N'$ or $'E'$, the Cholesky decomposition is used to factor the matrix A (after equilibration if $fact = 'E'$) as
$$A = U^T \cdot U \text{ (real)}, A = U^H \cdot U \text{ (complex), if } uplo = 'U',$$

$$\text{or } A = L \cdot L^T \text{ (real)}, A = L \cdot L^H \text{ (complex), if } uplo = 'L',$$

where U is an upper triangular band matrix and L is a lower triangular band matrix.
3. If the leading i -by- i principal minor is not positive definite, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
4. The system of equations is solved for X using the factored form of A .
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix X is premultiplied by $diag(s)$ so that it solves the original system before equilibration.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

fact

CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.

If $\text{fact} = \text{'F'}$: on entry, afb contains the factored form of A . If $\text{equed} = \text{'Y'}$, the matrix A has been equilibrated with scaling factors given by s .

ab and afb will not be modified.

If $\text{fact} = \text{'N'}$, the matrix A will be copied to afb and factored.

If $\text{fact} = \text{'E'}$, the matrix A will be equilibrated if necessary, then copied to afb and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored:

If $\text{uplo} = \text{'U'}$, the upper triangle of A is stored.

If $\text{uplo} = \text{'L'}$, the lower triangle of A is stored.

*n*INTEGER. The order of matrix A ; $n \geq 0$.*kd*INTEGER. The number of superdiagonals or subdiagonals in the matrix A ; $kd \geq 0$.*nrhs*INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.*ab, afb, b, work*

REAL for spbsvx

DOUBLE PRECISION for dpbsvx

COMPLEX for cpbsvx

DOUBLE COMPLEX for zpbsvx.

Arrays: $\text{ab}(ldab, *)$, $\text{afb}(ldab, *)$, $\text{b}(ldb, *)$, $\text{work}(*)$.

The array ab contains the upper or lower triangle of the matrix A in *band storage* (see [Matrix Storage Schemes](#)).

If $\text{fact} = \text{'F'}$ and $\text{equed} = \text{'Y'}$, then ab must contain the equilibrated matrix $\text{diag}(s) * A * \text{diag}(s)$. The second dimension of ab must be at least $\max(1, n)$.

The array afb is an input argument if $\text{fact} = \text{'F'}$. It contains the triangular factor U or L from the Cholesky factorization of the band matrix A in the same storage format as A . If $\text{equed} = \text{'Y'}$, then afb is the factored form of the equilibrated matrix A . The second dimension of afb must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$\text{work}(*)$ is a workspace array.

The dimension of work must be at least $\max(1, 3*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; $ldab \geq kd+1$.
<i>ldafb</i>	INTEGER. The leading dimension of <i>afb</i> ; $ldafb \geq kd+1$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>equed</i>	CHARACTER*1. Must be 'N' or 'Y'. <i>equed</i> is an input argument if <i>fact</i> = 'F'. It specifies the form of equilibration that was done: if <i>equed</i> = 'N', no equilibration was done (always true if <i>fact</i> = 'N') if <i>equed</i> = 'Y', equilibration was done, that is, <i>A</i> has been replaced by <i>diag(s)*A*diag(s)</i> .
<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION (<i>n</i>). The array <i>s</i> contains the scale factors for <i>A</i> . This array is an input argument if <i>fact</i> = 'F' only; otherwise it is an output argument. If <i>equed</i> = 'N', <i>s</i> is not accessed. If <i>fact</i> = 'F' and <i>equed</i> = 'Y', each element of <i>s</i> must be positive.
<i>ldx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $ldx \geq \max(1, n)$.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.
<i>rwork</i>	REAL for <i>cpbsvx</i> DOUBLE PRECISION for <i>zpbsvx</i> . Workspace array, DIMENSION at least $\max(1, n)$; used in complex flavors only.

Output Parameters

<i>x</i>	REAL for <i>spbsvx</i> DOUBLE PRECISION for <i>dpbsvx</i> COMPLEX for <i>cpbsvx</i> DOUBLE COMPLEX for <i>zpbsvx</i> . Array, DIMENSION (<i>lidx, *</i>). If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> to the original system of equations. Note that if <i>equed</i> = 'Y', <i>A</i> and <i>B</i> are modified on exit, and the solution to the equilibrated system is <i>inv(diag(s))*X</i> . The second dimension of <i>x</i> must be at least $\max(1, nrhs)$.
----------	---

<i>ab</i>	On exit, if <i>fact</i> = 'E' and <i>equed</i> = 'Y', <i>A</i> is overwritten by $\text{diag}(s) * A * \text{diag}(s)$.
<i>afb</i>	If <i>fact</i> = 'N' or 'E', then <i>afb</i> is an output argument and on exit returns the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ (real routines), $A = U^H * U$ or $A = L * L^H$ (complex routines) of the original matrix <i>A</i> (if <i>fact</i> = 'N'), or of the equilibrated matrix <i>A</i> (if <i>fact</i> = 'E'). See the description of <i>ab</i> for the form of the equilibrated matrix.
<i>b</i>	Overwritten by $\text{diag}(s) * B$, if <i>equed</i> = 'Y'; not changed if <i>equed</i> = 'N'.
<i>s</i>	This array is an output argument if <i>fact</i> ≠ 'F'. See the description of <i>s</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least <code>max(1, nrhs)</code> . Contains the estimated forward error bound for each solution vector $x(j)$ (the <i>j</i> -th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to $x(j)$, <i>ferr(j)</i> is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error.
<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least <code>max(1, nrhs)</code> . Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes $x(j)$ an exact solution.
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , and $i \leq n$, the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be

computed; $rcond = 0$ is returned. If $info = i$, and $i = n + 1$, then U is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbsvx` interface are as follows:

<code>ab</code>	Holds the array A of size $(kd+1, n)$.
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>x</code>	Holds the matrix X of size $(n, nrhs)$.
<code>afb</code>	Holds the array AF of size $(kd+1, n)$.
<code>s</code>	Holds the vector with the number of elements n . Default value for each element is $s(i) = 1.0_{WP}$.
<code>ferr</code>	Holds the vector with the number of elements $nrhs$.
<code>berr</code>	Holds the vector with the number of elements $nrhs$.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>fact</code>	Must be ' <code>N</code> ', ' <code>E</code> ', or ' <code>F</code> '. The default value is ' <code>N</code> '. If <code>fact = 'F'</code> , then <code>af</code> must be present; otherwise, an error is returned.
<code>equed</code>	Must be ' <code>N</code> ' or ' <code>Y</code> '. The default value is ' <code>N</code> '.

?ptsv

Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite tridiagonal matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sptsv( n, nrhs, d, e, b, ldb, info )
call dptsv( n, nrhs, d, e, b, ldb, info )
call cptsv( n, nrhs, d, e, b, ldb, info )
call zptsv( n, nrhs, d, e, b, ldb, info )
```

Fortran 95:

```
call ptsv( d, e, b [,info] )
```

C:

```

lapack_int LAPACKE_sptsv( int matrix_layout, lapack_int n, lapack_int nrhs, float* d,
float* e, float* b, lapack_int ldb );

lapack_int LAPACKE_dptsv( int matrix_layout, lapack_int n, lapack_int nrhs, double* d,
double* e, double* b, lapack_int ldb );

lapack_int LAPACKE_cptsv( int matrix_layout, lapack_int n, lapack_int nrhs, float* d,
lapack_complex_float* e, lapack_complex_float* b, lapack_int ldb );

lapack_int LAPACKE_zptsv( int matrix_layout, lapack_int n, lapack_int nrhs, double* d,
lapack_complex_double* e, lapack_complex_double* b, lapack_int ldb );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric/Hermitian positive-definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

A is factored as $A = L^*D^*L^T$ (real flavors) or $A = L^*D^*L^H$ (complex flavors), and the factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>d</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, dimension at least $\max(1, n)$. Contains the diagonal elements of the tridiagonal matrix A .
<i>e, b</i>	REAL for sptsv DOUBLE PRECISION for dptsv COMPLEX for cptsv DOUBLE COMPLEX for zptsv. Arrays: $e(n - 1)$, $b(ldb, *)$. The array e contains the $(n - 1)$ subdiagonal elements of A . The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

<i>d</i>	Overwritten by the n diagonal elements of the diagonal matrix D from the $L^*D^*L^T$ (real)/ $L^*D^*L^H$ (complex) factorization of A .
<i>e</i>	Overwritten by the $(n - 1)$ subdiagonal elements of the unit bidiagonal factor L from the factorization of A .
<i>b</i>	Overwritten by the solution matrix X .
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the solution has not been computed. The factorization has not been completed unless $i = n$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ptsvx` interface are as follows:

<i>d</i>	Holds the vector of length n .
<i>e</i>	Holds the vector of length $(n-1)$.
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.

?ptsvx

Uses factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite tridiagonal matrix A , and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call sptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work,
             info )
call dptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work,
             info )
call cptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work,
             rwork, info )
call zptsvx( fact, n, nrhs, d, e, df, ef, b, ldb, x, ldx, rcond, ferr, berr, work,
             rwork, info )
```

Fortran 95:

```
call ptsvx( d, e, b, x [,df] [,ef] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

C:

```

lapack_int LAPACKE_sptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs,
                           const float* d, const float* e, float* df, float* ef, const float* b, lapack_int ldb,
                           float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_dptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs,
                           const double* d, const double* e, double* df, double* ef, const double* b, lapack_int ldb,
                           double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

lapack_int LAPACKE_cptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs,
                           const float* d, const lapack_complex_float* e, float* df, lapack_complex_float* ef,
                           const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
                           float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs,
                           const double* d, const lapack_complex_double* e, double* df, lapack_complex_double* ef,
                           const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
                           double* rcond, double* ferr, double* berr );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the Cholesky factorization $A = L^*D^*L^T$ (real)/ $A = L^*D^*L^H$ (complex) to compute the solution to a real or complex system of linear equations $A^*X = B$, where A is a n -by- n symmetric or Hermitian positive definite tridiagonal matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ptsvx performs the following steps:

1. If $fact = 'N'$, the matrix A is factored as $A = L^*D^*L^T$ (real flavors)/ $A = L^*D^*L^H$ (complex flavors), where L is a unit lower bidiagonal matrix and D is diagonal. The factorization can also be regarded as having the form $A = U^T*D^*U$ (real flavors)/ $A = U^H*D^*U$ (complex flavors).
2. If the leading i -by- i principal minor is not positive-definite, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

fact

CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix A is supplied on entry.

If $fact = 'F'$: on entry, df and ef contain the factored form of A . Arrays d , e , df , and ef will not be modified.

If $\text{fact} = \text{'N'}$, the matrix A will be copied to df and ef , and factored.

n

INTEGER. The order of matrix A ; $n \geq 0$.

$nrhs$

INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.

$d, df, rwork$

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays: $d(n)$, $df(n)$, $rwork(n)$.

The array d contains the n diagonal elements of the tridiagonal matrix A .

The array df is an input argument if $\text{fact} = \text{'F'}$ and on entry contains the n diagonal elements of the diagonal matrix D from the L^*D*L^T (real)/ L^*D*L^H (complex) factorization of A .

The array $rwork$ is a workspace array used for complex flavors only.

$e, ef, b, work$

REAL for sptsvx

DOUBLE PRECISION for dptsvx

COMPLEX for cptsvx

DOUBLE COMPLEX for zptsvx.

Arrays: $e(n - 1)$, $ef(n - 1)$, $b(ldb*)$, $work(*)$. The array e contains the $(n - 1)$ subdiagonal elements of the tridiagonal matrix A .

The array ef is an input argument if $\text{fact} = \text{'F'}$ and on entry contains the $(n - 1)$ subdiagonal elements of the unit bidiagonal factor L from the L^*D*L^T (real)/ L^*D*L^H (complex) factorization of A .

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

The array $work$ is a workspace array. The dimension of $work$ must be at least $2*n$ for real flavors, and at least n for complex flavors.

ldb

INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

ldx

INTEGER. The leading dimension of x ; $ldx \geq \max(1, n)$.

Output Parameters

x

REAL for sptsvx

DOUBLE PRECISION for dptsvx

COMPLEX for cptsvx

DOUBLE COMPLEX for zptsvx.

Array, DIMENSION ($ldx, *$).

If $info = 0$ or $info = n+1$, the array x contains the solution matrix X to the system of equations. The second dimension of x must be at least $\max(1, nrhs)$.

<i>df, ef</i>	These arrays are output arguments if <i>fact</i> = 'N'. See the description of <i>df, ef</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. An estimate of the reciprocal condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least <code>max(1, nrhs)</code> . Contains the estimated forward error bound for each solution vector $x(j)$ (the <i>j</i> -th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to $x(j)$, <i>ferr(j)</i> is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error.
<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least <code>max(1, nrhs)</code> . Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes $x(j)$ an exact solution.
<i>info</i>	INTEGER . If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , and $i \leq n$, the leading minor of order <i>i</i> (and therefore the matrix <i>A</i> itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned. If <i>info</i> = <i>i</i> , and $i = n + 1$, then <i>U</i> is nonsingular, but <i>rcond</i> is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of <i>rcond</i> would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ptsvx` interface are as follows:

<i>d</i>	Holds the vector of length <i>n</i> .
----------	---------------------------------------

<i>e</i>	Holds the vector of length (<i>n</i> -1).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>x</i>	Holds the matrix <i>X</i> of size (<i>n</i> , <i>nrhs</i>).
<i>df</i>	Holds the vector of length <i>n</i> .
<i>ef</i>	Holds the vector of length (<i>n</i> -1).
<i>ferr</i>	Holds the vector of length (<i>nrhs</i>).
<i>berr</i>	Holds the vector of length (<i>nrhs</i>).
<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.

?sysv

Computes the solution to the system of linear equations with a real or complex symmetric matrix *A* and multiple right-hand sides.

Syntax**FORTRAN 77:**

```
call ssy whole( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call dsy whole( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call csy whole( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zsy whole( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
```

Fortran 95:

```
call sysv( a, b [,uplo] [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>sysv( int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, <datatype>* a, lapack_int lda, lapack_int* ipiv, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for *X* the real or complex system of linear equations $A \cdot X = B$, where *A* is an *n*-by-*n* symmetric matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions.

The diagonal pivoting method is used to factor *A* as $A = U \cdot D \cdot U^T$ or $A = L \cdot D \cdot L^T$, where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of *A* is then used to solve the system of equations $A \cdot X = B$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored:
	If <i>uplo</i> = 'U', the upper triangle of A is stored.
	If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in B ; $nrhs \geq 0$.
<i>a, b, work</i>	<p>REAL for ssyssv</p> <p>DOUBLE PRECISION for dsyssv</p> <p>COMPLEX for csyssv</p> <p>DOUBLE COMPLEX for zsyssv.</p> <p>Arrays: $a(1da, *)$, $b(1db, *)$, $work(*)$.</p> <p>The array a contains the upper or the lower triangular part of the symmetric matrix A (see <i>uplo</i>). The second dimension of a must be at least $\max(1, n)$.</p> <p>The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.</p> <p><i>work</i> is a workspace array, dimension at least $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of a ; $1da \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; $1db \geq \max(1, n)$.
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array; $lwork \geq 1$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i>. See <i>Application Notes</i> below for details and for the suggested value of <i>lwork</i>.</p>

Output Parameters

<i>a</i>	If <i>info</i> = 0, a is overwritten by the block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by ?sytrf .
<i>b</i>	If <i>info</i> = 0, b is overwritten by the solution matrix X .
<i>ipiv</i>	INTEGER.

Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of D , as determined by [?sytrf](#).

If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the i -th row and column of A was interchanged with the k -th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

$work(1)$

If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sysv` interface are as follows:

a Holds the matrix A of size (n, n) .

b Holds the matrix B of size $(n, nrhs)$.

$ipiv$ Holds the vector of length n .

$uplo$ Must be '`U`' or '`L`'. The default value is '`U`'.

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sysv_rook

Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix A and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call ssy whole ssy whole ssy whole ssy whole  
call dsy whole dsy whole dsy whole dsy whole  
call csy whole csy whole csy whole csy whole  
call zsy whole zsy whole zsy whole zsy whole
```

Fortran 95:

```
call sysv_rook( a, b [,uplo] [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>sysv_rook( int matrix_layout, char uplo, lapack_int n, lapack_int  
nrhs, <datatype>* a, lapack_int lda, lapack_int* ipiv, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the real or complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U^*D^*U^T$ or $A = L^*D^*L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The ?sysv_rook routine is called to compute the factorization of a complex symmetric matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If <i>uplo</i> = 'U', the upper triangle of A is stored. If <i>uplo</i> = 'L', the lower triangle of A is stored.

<i>n</i>	INTEGER. The order of matrix <i>A</i> ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in <i>B</i> ; $nrhs \geq 0$.
<i>a, b, work</i>	<p>REAL for <code>ssysv_rook</code> DOUBLE PRECISION for <code>dsssv_rook</code> COMPLEX for <code>csssv_rook</code> DOUBLE COMPLEX for <code>zsssv_rook</code>.</p> <p>Arrays: <i>a</i>(size <i>lda</i> by *), <i>b</i> (size <i>ldb</i> by *), <i>work</i>(*).</p>
	The array <i>a</i> contains the upper or the lower triangular part of the symmetric matrix <i>A</i> (see <i>uplo</i>). The second dimension of <i>a</i> must be at least $\max(1, n)$.
	The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$.
	<i>work</i> is a workspace array, dimension at least $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; $ldb \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; $lwork \geq 1$.
	If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code> . See <i>Application Notes</i> below for details and for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	If <i>info</i> = 0, <i>a</i> is overwritten by the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>) from the factorization of <i>A</i> as computed by ?sytrf_rook .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of <i>D</i>, as determined by ?sytrf_rook.</p> <p>If <i>ipiv</i>(<i>k</i>) > 0, then rows and columns <i>k</i> and <i>ipiv</i>(<i>k</i>) were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.</p> <p>If <i>uplo</i> = 'U' and <i>ipiv</i>(<i>k</i>) < 0 and <i>ipiv</i>(<i>k</i> - 1) < 0, then rows and columns <i>k</i> and -<i>ipiv</i>(<i>k</i>) were interchanged, rows and columns <i>k</i> - 1 and -<i>ipiv</i>(<i>k</i> - 1) were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.</p>

If $uplo = 'L'$ and $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns k and $-ipiv(k)$ were interchanged, rows and columns $k + 1$ and $-ipiv(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

<i>work(1)</i>	If $info = 0$, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<p>INTEGER. If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p> <p>If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.</p>

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sysv_rook` interface are as follows:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size $(n, nrhs)$.
<i>ipiv</i>	Holds the vector of length n .
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

See Also

[Matrix Storage Schemes](#)

?sysvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric matrix A , and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call ssysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, lwork, iwork, info )
call dsysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, lwork, iwork, info )
call csysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, lwork, rwork, info )
call zsysvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr,
berr, work, lwork, rwork, info )
```

Fortran 95:

```
call sysvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

C:

```

lapack_int LAPACKE_ssksvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const float* a, lapack_int lda, float* af, lapack_int ldaf,
lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int idx, float*
rcond, float* ferr, float* berr );

lapack_int LAPACKE_dsysvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const double* a, lapack_int lda, double* af, lapack_int ldat,
lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int idx, double*
rcond, double* ferr, double* berr );

lapack_int LAPACKE_csysvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, lapack_complex_float*
af, lapack_int ldat, lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int idx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zsysvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, lapack_complex_double*
af, lapack_int ldat, lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int idx, double* rcond, double* ferr, double* berr );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is a n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?sysvx performs the following steps:

1. If $fact = 'N'$, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U \cdot D \cdot U^T$ or $A = L \cdot D \cdot L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

fact

CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix A has been supplied on entry.

If $\text{fact} = \text{'F'}$: on entry, af and ipiv contain the factored form of A . Arrays a , af , and ipiv will not be modified.

If $\text{fact} = \text{'N'}$, the matrix A will be copied to af and factored.

uplo

CHARACTER*1. Must be ' U ' or ' L '.

Indicates whether the upper or lower triangular part of A is stored:

If $\text{uplo} = \text{'U'}$, the upper triangle of A is stored.

If $\text{uplo} = \text{'L'}$, the lower triangle of A is stored.

n

INTEGER. The order of matrix A ; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides, the number of columns in B ; $\text{nrhs} \geq 0$.

$a, \text{af}, b, \text{work}$

REAL for `ssysvx`

DOUBLE PRECISION for `dsysvx`

COMPLEX for `csysvx`

DOUBLE COMPLEX for `zsysvx`.

Arrays: $a(\text{lda}, *), \text{af}(\text{ldaf}, *), b(\text{ldb}, *), \text{work}(*)$.

The array a contains the upper or the lower triangular part of the symmetric matrix A (see uplo). The second dimension of a must be at least $\max(1, n)$.

The array af is an input argument if $\text{fact} = \text{'F'}$. It contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U^*D^*U^{**}T$ or $A = L^*D^*L^{**}T$ as computed by [?sytrf](#). The second dimension of af must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, \text{nrhs})$.

$\text{work}(*)$ is a workspace array, dimension at least $\max(1, \text{lwork})$.

lda

INTEGER. The leading dimension of a ; $\text{lda} \geq \max(1, n)$.

ldaf

INTEGER. The leading dimension of af ; $\text{ldaf} \geq \max(1, n)$.

ldb

INTEGER. The leading dimension of b ; $\text{ldb} \geq \max(1, n)$.

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array ipiv is an input argument if $\text{fact} = \text{'F'}$. It contains details of the interchanges and the block structure of D , as determined by [?sytrf](#).

If $\text{ipiv}(i) = k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the i -th row and column of A was interchanged with the k -th row and column.

If $\text{uplo} = \text{'U'}$ and $\text{ipiv}(i) = \text{ipiv}(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

ldx

INTEGER. The leading dimension of the output array x ; $ldx \geq \max(1, n)$.

lwork

INTEGER. The size of the *work* array.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*. See *Application Notes* below for details and for the suggested value of *lwork*.

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.

rwork

REAL for *csysvx*;
DOUBLE PRECISION for *zsysvx*.
Workspace array, DIMENSION at least $\max(1, n)$; used in complex flavors only.

Output Parameters

x

REAL for *ssysvx*
DOUBLE PRECISION for *dsysvx*
COMPLEX for *csysvx*
DOUBLE COMPLEX for *zsysvx*.
Array, DIMENSION (*ldx*, *).

If $info = 0$ or $info = n+1$, the array x contains the solution matrix X to the system of equations. The second dimension of x must be at least $\max(1, nrhs)$.

af, ipiv

These arrays are output arguments if $fact = 'N'$.

See the description of *af, ipiv* in *Input Arguments* section.

rcond

REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix A . If *rcond* is less than the machine precision (in particular, if *rcond* = 0), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr

REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j -th column of the solution matrix X). If *xtrue* is the true solution corresponding to

$x(j)$, $ferr(j)$ is an estimated upper bound for the magnitude of the largest element in $(x(j) - x_{true})$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.

berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

work(1)

If $info=0$, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

*info*INTEGER. If $info = 0$, the execution is successful.If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n + 1$, then D is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sysvx` interface are as follows:

*a*Holds the matrix A of size (n, n) .*b*Holds the matrix B of size $(n, nrhs)$.*x*Holds the matrix X of size $(n, nrhs)$.*af*Holds the matrix AF of size (n, n) .*ipiv*Holds the vector of length n .*ferr*Holds the vector of length $(nrhs)$.*berr*Holds the vector of length $(nrhs)$.*uplo*Must be '`U`' or '`L`'. The default value is '`U`'.

<i>fact</i>	Must be 'N' or 'F'. The default value is 'N'. If <i>fact</i> = 'F', then both arguments <i>af</i> and <i>ipiv</i> must be present; otherwise, an error is returned.
-------------	---

Application Notes

The value of *lwork* must be at least $\max(1, m \cdot n)$, where for real flavors $m = 3$ and for complex flavors $m = 2$. For better performance, try using $lwork = \max(1, m \cdot n, n \cdot \text{blocksize})$, where *blocksize* is the optimal block size for *?sytrf*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sysvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a symmetric indefinite matrix *A* applying the diagonal pivoting factorization.

Syntax

FORTRAN 77:

```
call ssy whole fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
      rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
      iwork, info )

call dsy whole fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
      rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
      iwork, info )

call csy whole fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
      rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
      rwork, info )

call zsy whole fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, ldx,
      rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
      rwork, info )
```

C:

```
lapack_int LAPACKE_ssypvxx( int matrix_layout, char fact, char uplo, lapack_int n,
                            lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, lapack_int* ipiv, char* equed, float* s, float* b, lapack_int ldb, float* x, lapack_int ldx,
                            float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm,
                            float* err_bnds_comp, lapack_int nparams, const float* params );
```

```

lapack_int LAPACKE_dsysvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, lapack_int* ipiv, char* equed, double* s, double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double* params );

lapack_int LAPACKE_csysvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, float* s, lapack_complex_float* b,
lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* rpvgrw,
float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp,
lapack_int nparams, const float* params );

lapack_int LAPACKE_zsysvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, double* s, lapack_complex_double* b,
lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double* rpvgrw,
double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp,
lapack_int nparams, const double* params );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the *diagonal pivoting* factorization $A=U^T \cdot U$ (real flavors) / $A=U^H \cdot U$ (complex flavors) or $A=L \cdot L^T$ (real flavors) / $A=L \cdot L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is an n -by- n real symmetric/Hermitian matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($\mathcal{O}(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $\mathcal{O}(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?sysvxx performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

$$\text{diag}(s) \cdot A \cdot \text{diag}(s) \cdot \text{inv}(\text{diag}(s)) \cdot X = \text{diag}(s) \cdot B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(s) \cdot A \cdot \text{diag}(s)$ and B by $\text{diag}(s) \cdot B$.

2. If *fact* = 'N' or 'E', the LU decomposition is used to factor the matrix A (after equilibration if *fact* = 'E') as

$$A = U \cdot D \cdot U^T, \text{ if } \text{uplo} = 'U',$$

$$\text{or } A = L \cdot D \cdot L^T, \text{ if } \text{uplo} = 'L',$$

where U or L is a product of permutation and unit upper (lower) triangular matrices, and D is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

3. If some $D(i, i) = 0$, so that D is exactly singular, the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A (see the $rcond$ parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for X and compute error bounds.
4. The system of equations is solved for X using the factored form of A .
5. By default, unless `params(la_linx_itref_i)` is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix X is premultiplied by $diag(r)$ so that it solves the original system before equilibration.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>fact</i>	CHARACTER*1. Must be 'F', 'N', or 'E'.
	Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.
	If <i>fact</i> = 'F', on entry, <i>af</i> and <i>ipiv</i> contain the factored form of A . If <i>equed</i> is not 'N', the matrix A has been equilibrated with scaling factors given by <i>s</i> . Parameters <i>a</i> , <i>af</i> , and <i>ipiv</i> are not modified.
	If <i>fact</i> = 'N', the matrix A will be copied to <i>af</i> and factored.
	If <i>fact</i> = 'E', the matrix A will be equilibrated, if necessary, copied to <i>af</i> and factored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored:
	If <i>uplo</i> = 'U', the upper triangle of A is stored.
	If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.
<i>a, af, b, work</i>	REAL for ssyvxx DOUBLE PRECISION for dsyvxx COMPLEX for csyvxx DOUBLE COMPLEX for zsyvxx. Arrays: <i>a</i> (<i>lda</i> , *), <i>af</i> (<i>ldaf</i> , *), <i>b</i> (<i>ldb</i> , *), <i>work</i> (*).
	The array <i>a</i> contains the symmetric matrix A as specified by <i>uplo</i> . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of <i>a</i> contains the upper triangular part of the matrix A and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of <i>a</i> contains the lower triangular part of the matrix A and the strictly upper triangular part of <i>a</i> is not referenced. The second dimension of <i>a</i> must be at least $\max(1, n)$.

The array af is an input argument if $fact = 'F'$. It contains the block diagonal matrix D and the multipliers used to obtain the factor U and L from the factorization $A = U^* D^* U^{**} T$ or $A = L^* D^* L^{**} T$ as computed by [?sytrf](#). The second dimension of af must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 4*n)$ for real flavors, and at least $\max(1, 2*n)$ for complex flavors.

lda

INTEGER. The leading dimension of the array a ; $lda \geq \max(1, n)$.

$ldaf$

INTEGER. The leading dimension of the array af ; $ldaf \geq \max(1, n)$.

$ipiv$

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array $ipiv$ is an input argument if $fact = 'F'$. It contains details of the interchanges and the block structure of D as determined by [?sytrf](#). If $ipiv(k) > 0$, rows and columns k and $ipiv(k)$ were interchanged and $D(k,k)$ is a 1-by-1 diagonal block.

If $ipiv = 'U'$ and $ipiv(k) = ipiv(k-1) < 0$, rows and columns $k-1$ and $-ipiv(k)$ were interchanged and $D(k-1:k, k-1:k)$ is a 2-by-2 diagonal block.

If $ipiv = 'L'$ and $ipiv(k) = ipiv(k+1) < 0$, rows and columns $k+1$ and $-ipiv(k)$ were interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.

$equed$

CHARACTER*1. Must be ' N ' or ' Y '.

$equed$ is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:

If $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$).

If $equed = 'Y'$, both row and column equilibration was done, that is, A has been replaced by $\text{diag}(s)^* A^* \text{diag}(s)$.

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (n). The array s contains the scale factors for A . If $equed = 'Y'$, A is multiplied on the left and right by $\text{diag}(s)$.

This array is an input argument if $fact = 'F'$ only; otherwise it is an output argument.

If $fact = 'F'$ and $equed = 'Y'$, each element of s must be positive.

Each element of s should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

<i>lidx</i>	INTEGER. The leading dimension of the output array <i>x</i> ; $lidx \geq \max(1, n)$.								
<i>n_err_bnds</i>	INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See <i>err_bnds_norm</i> and <i>err_bnds_comp</i> descriptions in the <i>Output Arguments</i> section below.								
<i>nparams</i>	INTEGER. Specifies the number of parameters set in <i>params</i> . If ≤ 0 , the <i>params</i> array is never referenced and default values are used.								
<i>params</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Array, DIMENSION <i>nparams</i>. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to <i>nparams</i> are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass <i>nparams</i> = 0, which prevents the source code from accessing the <i>params</i> argument.</p> <p><i>params(la_linx_itref_i = 1)</i> : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).</p> <table> <tr> <td>=0.0</td><td>No refinement is performed and no error bounds are computed.</td></tr> <tr> <td>=1.0</td><td>Use the extra-precise refinement algorithm.</td></tr> </table> <p>(Other values are reserved for future use.)</p> <p><i>params(la_linx_ithresh_i = 2)</i> : Maximum number of residual computations allowed for refinement.</p> <table> <tr> <td>Default</td><td>10</td></tr> <tr> <td>Aggressive</td><td>Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.</td></tr> </table> <p><i>params(la_linx_cwise_i = 3)</i> : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).</p>	=0.0	No refinement is performed and no error bounds are computed.	=1.0	Use the extra-precise refinement algorithm.	Default	10	Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.
=0.0	No refinement is performed and no error bounds are computed.								
=1.0	Use the extra-precise refinement algorithm.								
Default	10								
Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>err_bnds_norm</i> and <i>err_bnds_comp</i> may no longer be trustworthy.								
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.								
<i>rwork</i>	<p>REAL for single precision flavors</p> <p>DOUBLE PRECISION for double precision flavors.</p> <p>Workspace array, DIMENSION at least $\max(1, 3*n)$; used in complex flavors only.</p>								

Output Parameters

<i>x</i>	REAL for ssyvxx DOUBLE PRECISION for dsyvxx COMPLEX for csyvxx DOUBLE COMPLEX for zsyvxx. Array, DIMENSION (<i>ldx</i> , <i>nrhs</i>). If <i>info</i> = 0, the array <i>x</i> contains the solution <i>n</i> -by- <i>nrhs</i> matrix <i>X</i> to the original system of equations. Note that <i>A</i> and <i>B</i> are modified on exit if <i>equed</i> ≠ 'N', and the solution to the equilibrated system is: <i>inv(diag(s)) * X.</i>
<i>a</i>	If <i>fact</i> = 'E' and <i>equed</i> = 'Y', overwritten by <i>diag(s) * A * diag(s)</i> .
<i>af</i>	If <i>fact</i> = 'N', <i>af</i> is an output argument and on exit returns the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> from the factorization <i>A</i> = <i>U</i> * <i>D</i> * <i>U</i> ** <i>T</i> or <i>A</i> = <i>L</i> * <i>D</i> * <i>L</i> ** <i>T</i> .
<i>b</i>	If <i>equed</i> = 'N', <i>B</i> is not modified. If <i>equed</i> = 'Y', <i>B</i> is overwritten by <i>diag(s) * B</i> .
<i>s</i>	This array is an output argument if <i>fact</i> ≠ 'F'. Each element of this array is a power of the radix. See the description of <i>s</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision, in particular, if <i>rcond</i> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.
<i>rpvgrw</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Contains the reciprocal pivot growth factor <i>norm(A) / norm(U)</i> . The <i>max absolute element</i> norm is used. If this is much less than 1, the stability of the <i>LU</i> factorization of the (equilibrated) matrix <i>A</i> could be poor. This also means that the solution <i>X</i> , estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < \text{info} \leq n$, this parameter contains the reciprocal pivot growth factor for the leading <i>info</i> columns of <i>A</i> .
<i>berr</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least <i>max(1, nrhs)</i> . Contains the componentwise relative backward error for each solution vector <i>x(j)</i> , that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes <i>x(j)</i> an exact solution.

`err_bnds_norm`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (*nrhs, n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i, :)` corresponds to the *i*-th right-hand side.

The second index in `err_bnds_norm(:, err)` contains the following three fields:

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.

`err=2`

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix Z.

Let $z = s * a$, where *s* scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

`Array, DIMENSION (nrhs, n_err_bnds)`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3) = 0.0*), then *err_bnds_comp* is not accessed. If *n_err_bnds < 3*, then at most the first $(:, n_err_bnds)$ entries are returned.

The first index in *err_bnds_comp(i, :)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_comp(:, err)* contains the following three fields:

<i>err=1</i>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.
<i>err=2</i>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<i>err=3</i>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix <i>Z</i> . Let $z = s * (a * \text{diag}(x))$, where <i>x</i> is the solution for the current right-hand side and <i>s</i> scales each row of <i>a * diag(x)</i> by a power of the radix so all absolute row sums of <i>z</i> are approximately 1.

<i>ipiv</i>	If <i>fact</i> = 'N', <i>ipiv</i> is an output argument and on exit contains details of the interchanges and the block structure <i>D</i> , as determined by ssytrf for single precision flavors and dsytrf for double precision flavors.
<i>equed</i>	If <i>fact</i> ≠ 'F', then <i>equed</i> is an output argument. It specifies the form of equilibration that was done (see the description of <i>equed</i> in <i>Input Arguments</i> section).
<i>params</i>	If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. The solution to every right-hand side is guaranteed. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If $0 < \text{info} \leq n$: <i>U</i> (<i>info</i> , <i>info</i>) is exactly zero. The factorization has been completed, but the factor <i>U</i> is exactly singular, so the solution and error bounds could not be computed; <i>rcond</i> = 0 is returned. If <i>info</i> = <i>n+j</i> : The solution corresponding to the <i>j</i> -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides <i>k</i> with <i>k</i> > <i>j</i> may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested <i>params</i> (3) = 0.0, then the <i>j</i> -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest <i>j</i> such that <i>err_bnds_norm(j,1)</i> = 0.0 or <i>err_bnds_comp(j,1)</i> = 0.0. See the definition of <i>err_bnds_norm(;;1)</i> and <i>err_bnds_comp(;;1)</i> . To get information about all of the right-hand sides, check <i>err_bnds_norm</i> or <i>err_bnds_comp</i> .

?hesv

Computes the solution to the system of linear equations with a Hermitian matrix *A* and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call chesv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zhesv( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
```

Fortran 95:

```
call hesv( a, b [,uplo] [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>hesv( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, <datatype>* a, lapack_int* lda, lapack_int* ipiv, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the complex system of linear equations $A^*X = B$, where A is an n -by- n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U^*D^*U^H$ or $A = L^*D^*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix A , and A is factored as $U^*D^*U^H$.

If *uplo* = 'L', the array *a* stores the lower triangular part of the matrix A , and A is factored as $L^*D^*L^H$.

n

INTEGER. The order of matrix A ; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.

a, b, work

COMPLEX for chesv

DOUBLE COMPLEX for zhesv.

Arrays: *a*(*lda*, *), *b*(*ldb*, *), *work*(*). The array *a* contains the upper or the lower triangular part of the Hermitian matrix A (see *uplo*). The second dimension of *a* must be at least $\max(1, n)$.

The array *b* contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of *b* must be at least $\max(1, nrhs)$.

work is a workspace array, dimension at least $\max(1, lwork)$.

lda

INTEGER. The leading dimension of *a*; *lda* $\geq \max(1, n)$.

ldb

INTEGER. The leading dimension of *b*; *ldb* $\geq \max(1, n)$.

lwork

INTEGER. The size of the *work* array (*lwork* ≥ 1).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*. See *Application Notes* below for details and for the suggested value of *lwork*.

Output Parameters

<i>a</i>	If <i>info</i> = 0, <i>a</i> is overwritten by the block-diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> (or <i>L</i>) from the factorization of <i>A</i> as computed by ?hetrf .
<i>b</i>	If <i>info</i> = 0, <i>b</i> is overwritten by the solution matrix <i>X</i> .
<i>ipiv</i>	INTEGER. Array, DIMENSION at least <code>max(1, n)</code> . Contains details of the interchanges and the block structure of <i>D</i> , as determined by ?hetrf . If <i>ipiv</i> (<i>i</i>) = <i>k</i> > 0, then <i>d_{ii}</i> is a 1-by-1 diagonal block, and the <i>i</i> -th row and column of <i>A</i> was interchanged with the <i>k</i> -th row and column. If <i>uplo</i> = 'U' and <i>ipiv</i> (<i>i</i>) = <i>ipiv</i> (<i>i</i> -1) = - <i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> -1, and (<i>i</i> -1)-th row and column of <i>A</i> was interchanged with the <i>m</i> -th row and column. If <i>uplo</i> = 'L' and <i>ipiv</i> (<i>i</i>) = <i>ipiv</i> (<i>i</i> +1) = - <i>m</i> < 0, then <i>D</i> has a 2-by-2 block in rows/columns <i>i</i> and <i>i</i> +1, and (<i>i</i> +1)-th row and column of <i>A</i> was interchanged with the <i>m</i> -th row and column.
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , <i>d_{ii}</i> is 0. The factorization has been completed, but <i>D</i> is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hesv` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>nrhs</i>).
<i>ipiv</i>	Holds the vector of length <i>n</i> .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hesv_rook

Computes the solution to the system of linear equations for Hermitian matrices using the bounded Bunch-Kaufman diagonal pivoting method.

Syntax

FORTRAN 77:

```
call chesv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
call zhesv_rook( uplo, n, nrhs, a, lda, ipiv, b, ldb, work, lwork, info )
```

Fortran 95:

```
call hesv_rook( a, b [,uplo] [,ipiv] [,info] )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for *X* the complex system of linear equations $A^*X = B$, where *A* is an *n*-by-*n* Hermitian matrix, and *X* and *B* are *n*-by-*nrhs* matrices.

The bounded Bunch-Kaufman ("rook") diagonal pivoting method is used to factor *A* as

$A = U^*D^*U^H$ if *uplo* = 'U', or

$A = L^*D^*L^H$ if *uplo* = 'L',

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

?hetrf_rook is called to compute the factorization of a complex Hermitian matrix *A* using the bounded Bunch-Kaufman ("rook") diagonal pivoting method.

The factored form of *A* is then used to solve the system of equations $A^*X = B$ by calling ?HETRS_ROOK, which uses BLAS level 2 routines.

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of *A* is stored:

If *uplo* = 'U', the array *a* stores the upper triangular part of the matrix *A*.

If $uplo = 'L'$, the array a stores the lower triangular part of the matrix A .

n

INTEGER. The number of linear equations, which is the order of matrix A ; $n \geq 0$.

nrhs

INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.

a, b, work

COMPLEX for `chesv_rook`

COMPLEX*16 for `zhesv_rook`.

Arrays: a (size lda by *), b (size ldb by *), $work(*)$.

The array a contains the Hermitian matrix A . If $uplo = 'U'$, the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced. The second dimension of a must be at least $\max(1, n)$.

The array b contains the n -by- $nrhs$ right hand side matrix B . The second dimension of b must be at least $\max(1, nrhs)$.

$work$ is a workspace array, dimension at least $\max(1, lwork)$.

lda

INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldb

INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

lwork

INTEGER. The size of the $work$ array ($lwork \geq 1$).

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`. See *Application Notes* below for details and for the suggested value of $lwork$.

Output Parameters

a

If $info = 0$, a is overwritten by the block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by `?hetrf_rook`.

b

If $info = 0$, b is overwritten by the n -by- $nrhs$ solution matrix X .

ipiv

INTEGER.

Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of D , as determined by `?hetrf_rook`.

- If $uplo = 'U'$:

If $ipiv(k) > 0$, rows and columns k and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, rows and columns k and $-ipiv(k)$ were interchanged, rows and columns $k - 1$ and $-ipiv(k - 1)$ were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.

- If $uplo = 'L'$:

If $ipiv(k) > 0$, rows and columns k and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, rows and columns k and $-ipiv(k)$ were interchanged, rows and columns $k + 1$ and $-ipiv(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

$work(1)$

If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, D_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hesv_rook` interface are as follows:

a Holds the matrix A of size (n, n) .

b Holds the matrix B of size $(n, nrhs)$.

$ipiv$ Holds the vector of length n .

$uplo$ Must be '`U`' or '`L`'. The default value is '`U`'.

See Also

[Matrix Storage Schemes](#)

?hesvx

Uses the diagonal pivoting factorization to compute the solution to the complex system of linear equations with a Hermitian matrix A , and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call chesvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr,
            berr, work, lwork, rwork, info )
```

```
call zhesvx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, b, ldb, x, ldx, rcond, ferr,
            berr, work, lwork, rwork, info )
```

Fortran 95:

```
call hesvx( a, b, x [,uplo] [,af] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

C:

```
lapack_int LAPACKE_chesvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, lapack_complex_float*
af, lapack_int ldaf, lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zhesvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, lapack_complex_double*
af, lapack_int ldaf, lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $A^*X = B$, where A is an n -by- n Hermitian matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?hesvx performs the following steps:

1. If $fact = 'N'$, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U^*D^*U^H$ or $A = L^*D^*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

fact

CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix A has been supplied on entry.

If $fact = 'F'$: on entry, *af* and *ipiv* contain the factored form of A . Arrays *a*, *af*, and *ipiv* are not modified.

If $fact = 'N'$, the matrix A is copied to *af* and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If $uplo = 'U'$, the array a stores the upper triangular part of the Hermitian matrix A , and A is factored as $U^*D^*U^H$.

If $uplo = 'L'$, the array a stores the lower triangular part of the Hermitian matrix A ; A is factored as $L^*D^*L^H$.

n

INTEGER. The order of matrix A ; $n \geq 0$.

$nrhs$

INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.

$a, af, b, work$

COMPLEX for chesvx

DOUBLE COMPLEX for zhesvx.

Arrays: $a(1da, *), af(1daf, *), b(1db, *), work(*)$.

The array a contains the upper or the lower triangular part of the Hermitian matrix A (see $uplo$). The second dimension of a must be at least $\max(1, n)$.

The array af is an input argument if $fact = 'F'$. It contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U^*D^*U^H$ or $A = L^*D^*L^H$ as computed by [?hetrf](#). The second dimension of af must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

$work(*)$ is a workspace array of dimension at least $\max(1, lwork)$.

$1da$

INTEGER. The leading dimension of a ; $1da \geq \max(1, n)$.

$1daf$

INTEGER. The leading dimension of af ; $1daf \geq \max(1, n)$.

$1db$

INTEGER. The leading dimension of b ; $1db \geq \max(1, n)$.

$ipiv$

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array $ipiv$ is an input argument if $fact = 'F'$. It contains details of the interchanges and the block structure of D , as determined by [?hetrf](#).

If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the i -th row and column of A was interchanged with the k -th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

$1dx$

INTEGER. The leading dimension of the output array x ; $1dx \geq \max(1, n)$.

<i>lwork</i>	INTEGER. The size of the <i>work</i> array. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> below for details and for the suggested value of <i>lwork</i> .
<i>rwork</i>	REAL for <i>chesvx</i> DOUBLE PRECISION for <i>zhesvx</i> . Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>x</i>	COMPLEX for <i>chesvx</i> DOUBLE COMPLEX for <i>zhesvx</i> . Array, DIMENSION (<i>lidx</i> , *). If <i>info</i> = 0 or <i>info</i> = <i>n</i> +1, the array <i>x</i> contains the solution matrix <i>X</i> to the system of equations. The second dimension of <i>x</i> must be at least $\max(1, \text{nrhs})$.
<i>af</i> , <i>ipiv</i>	These arrays are output arguments if <i>fact</i> = 'N'. See the description of <i>af</i> , <i>ipiv</i> in <i>Input Arguments</i> section.
<i>rcond</i>	REAL for <i>chesvx</i> DOUBLE PRECISION for <i>zhesvx</i> . An estimate of the reciprocal condition number of the matrix <i>A</i> . If <i>rcond</i> is less than the machine precision (in particular, if <i>rcond</i> = 0), the matrix is singular to working precision. This condition is indicated by a return code of <i>info</i> > 0.
<i>ferr</i>	REAL for <i>chesvx</i> DOUBLE PRECISION for <i>zhesvx</i> . Array, DIMENSION at least $\max(1, \text{nrhs})$. Contains the estimated forward error bound for each solution vector <i>x(j)</i> (the <i>j</i> -th column of the solution matrix <i>X</i>). If <i>xtrue</i> is the true solution corresponding to <i>x(j)</i> , <i>ferr(j)</i> is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in <i>x(j)</i> . The estimate is as reliable as the estimate for <i>rcon</i> , and is almost always a slight overestimate of the true error.
<i>berr</i>	REAL for <i>chesvx</i> DOUBLE PRECISION for <i>zhesvx</i> . Array, DIMENSION at least $\max(1, \text{nrhs})$. Contains the component-wise relative backward error for each solution vector <i>x(j)</i> , that is, the smallest relative change in any element of <i>A</i> or <i>B</i> that makes <i>x(j)</i> an exact solution.

<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	<p>INTEGER. If <code>info = 0</code>, the execution is successful.</p> <p>If <code>info = -i</code>, the i-th parameter had an illegal value.</p> <p>If <code>info = i</code>, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution and error bounds could not be computed; <code>rcond = 0</code> is returned.</p> <p>If <code>info = i</code>, and $i = n + 1$, then D is nonsingular, but <code>rcond</code> is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of <code>rcond</code> would suggest.</p>

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hesvx` interface are as follows:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size $(n, nrhs)$.
<code>x</code>	Holds the matrix X of size $(n, nrhs)$.
<code>af</code>	Holds the matrix AF of size (n, n) .
<code>ipiv</code>	Holds the vector of length n .
<code>ferr</code>	Holds the vector of length $(nrhs)$.
<code>berr</code>	Holds the vector of length $(nrhs)$.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>fact</code>	Must be ' <code>N</code> ' or ' <code>F</code> '. The default value is ' <code>N</code> '. If <code>fact = 'F'</code> , then both arguments <code>af</code> and <code>ipiv</code> must be present; otherwise, an error is returned.

Application Notes

The value of `lwork` must be at least $2 * n$. For better performance, try using `lwork = n * blocksize`, where `blocksize` is the optimal block size for `?hetrf`.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hesvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a Hermitian indefinite coefficient matrix A applying the diagonal pivoting factorization.

Syntax

FORTRAN 77:

```
call chesvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, idx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
call zhesvxx( fact, uplo, n, nrhs, a, lda, af, ldaf, ipiv, equed, s, b, ldb, x, idx,
rcond, rpvgrw, berr, n_err_bnds, err_bnds_norm, err_bnds_comp, nparams, params, work,
rwork, info )
```

C:

```
lapack_int LAPACKE_chesvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, float* s, lapack_complex_float* b,
lapack_int ldb, lapack_complex_float* x, lapack_int idx, float* rcond, float* rpvgrw,
float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp,
lapack_int nparams, const float* params );
lapack_int LAPACKE_zhesvxx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, lapack_int* ipiv, char* equed, double* s, lapack_complex_double* b,
lapack_int ldb, lapack_complex_double* x, lapack_int idx, double* rcond, double*
rpvgrw, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double*
err_bnds_comp, lapack_int nparams, const double* params );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the *diagonal pivoting* factorization to compute the solution to a complex/double complex system of linear equations $A^*X = B$, where A is an n -by- n Hermitian matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\text{eps})$, where eps is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the *fact* and *equed* options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\text{eps})$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?hesvxx performs the following steps:

1. If *fact* = 'E', scaling factors are computed to equilibrate the system:

$$\text{diag}(s) * A * \text{diag}(s) * \text{inv}(\text{diag}(s)) * X = \text{diag}(s) * B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(s)*A*diag(s)* and *B* by *diag(s)*B*.

2. If *fact* = 'N' or 'E', the LU decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as

$$A = U * D * U^T, \text{ if } \text{uplo} = 'U',$$

$$\text{or } A = L * D * L^T, \text{ if } \text{uplo} = 'L',$$

where *U* or *L* is a product of permutation and unit upper (lower) triangular matrices, and *D* is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

3. If some $D(i,i)=0$, so that *D* is exactly singular, the routine returns with *info* = *i*. Otherwise, the factored form of *A* is used to estimate the condition number of the matrix *A* (see the *rcond* parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for *X* and compute error bounds.
4. The system of equations is solved for *X* using the factored form of *A*.
5. By default, unless *params(1)* is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.
6. If equilibration was used, the matrix *X* is premultiplied by *diag(r)* so that it solves the original system before equilibration.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>fact</i>	CHARACTER*1. Must be 'F', 'N', or 'E'.
	Specifies whether or not the factored form of the matrix <i>A</i> is supplied on entry, and if not, whether the matrix <i>A</i> should be equilibrated before it is factored.
	If <i>fact</i> = 'F', on entry, <i>af</i> and <i>ipiv</i> contain the factored form of <i>A</i> . If <i>equed</i> is not 'N', the matrix <i>A</i> has been equilibrated with scaling factors given by <i>s</i> . Parameters <i>a</i> , <i>af</i> , and <i>ipiv</i> are not modified.
	If <i>fact</i> = 'N', the matrix <i>A</i> will be copied to <i>af</i> and factored.
	If <i>fact</i> = 'E', the matrix <i>A</i> will be equilibrated, if necessary, copied to <i>af</i> and factored.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of A is stored: If $uplo = 'U'$, the upper triangle of A is stored. If $uplo = 'L'$, the lower triangle of A is stored.
<i>n</i>	INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the matrices B and X ; $nrhs \geq 0$.
<i>a, af, b, work</i>	COMPLEX for chesvxx DOUBLE COMPLEX for zhesvxx. Arrays: a (size lda by *), af (size $ldaf$ by *), b (ldb , *), $work(*)$. The array a contains the Hermitian matrix A as specified by <i>uplo</i> . If $uplo = 'U'$, the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A and the strictly lower triangular part of a is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A and the strictly upper triangular part of a is not referenced. The second dimension of a must be at least $\max(1, n)$. The array af is an input argument if <i>fact</i> = 'F'. It contains the block diagonal matrix D and the multipliers used to obtain the factor U and L from the factorization $A = U^* D^* U^T$ or $A = L^* D^* L^T$ as computed by ?hetrf . The second dimension of af must be at least $\max(1, n)$. The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$. $work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 5*n)$. The leading dimension of the array a ; $lda \geq \max(1, n)$. The leading dimension of the array af ; $ldaf \geq \max(1, n)$. INTEGER. Array, size at least $\max(1, n)$. The array <i>ipiv</i> is an input argument if <i>fact</i> = 'F'. It contains details of the interchanges and the block structure of D as determined by ?sytrf . If $ipiv(k) > 0$, rows and columns k and $ipiv(k)$ were interchanged and $D(k,k)$ is a 1-by-1 diagonal block. If $uplo = 'U'$ and $ipiv(k) = ipiv(k-1) < 0$, rows and columns $k-1$ and $-ipiv(k)$ were interchanged and $D(k-1:k, k-1:k)$ is a 2-by-2 diagonal block. If $uplo = 'L'$ and $ipiv(k) = ipiv(k+1) < 0$, rows and columns $k+1$ and $-ipiv(k)$ were interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.
<i>equed</i>	CHARACTER*1. Must be 'N' or 'Y'.

equed is an input argument if *fact* = 'F'. It specifies the form of equilibration that was done:

If *equed* = 'N', no equilibration was done (always true if *fact* = 'N').

If *equed* = 'Y', both row and column equilibration was done, that is, *A* has been replaced by *diag(s)*A*diag(s)*.

s

REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Array, size (*n*). The array *s* contains the scale factors for *A*. If *equed* = 'Y', *A* is multiplied on the left and right by *diag(s)*.

This array is an input argument if *fact* = 'F' only; otherwise it is an output argument.

If *fact* = 'F' and *equed* = 'Y', each element of *s* must be positive.

Each element of *s* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array *b*; *ldb* $\geq \max(1, n)$.

ldx

INTEGER. The leading dimension of the output array *x*; *ldx* $\geq \max(1, n)$.

n_err_bnds

INTEGER. Number of error bounds to return for each right hand side and each type (normwise or componentwise). See *err_bnds_norm* and *err_bnds_comp* descriptions in the *Output Arguments* section below.

nparams

INTEGER. Specifies the number of parameters set in *params*. If ≤ 0 , the *params* array is never referenced and default values are used.

params

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, size *nparams*. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to *nparams* are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass *nparams* = 0, which prevents the source code from accessing the *params* argument.

params(1) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0	No refinement is performed and no error bounds are computed.
------	--

=1.0	Use the extra-precise refinement algorithm.
------	---

(Other values are reserved for future use.)

params(2) : Maximum number of residual computations allowed for refinement.

Default	10
Aggressive	Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in <code>err_bnds_norm</code> and <code>err_bnds_comp</code> may no longer be trustworthy.

`params(3)` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

`rwork`

REAL for `chesvxx`

DOUBLE PRECISION for `zhesvxx`.

Workspace array, size at least `max(1, 3*n)`; used in complex flavors only.

Output Parameters

`x`

COMPLEX for `chesvxx`

DOUBLE COMPLEX for `zhesvxx`.

Array, size `lidx` by `nrhs`.

If `info = 0`, the array `x` contains the solution n -by- $nrhs$ matrix X to the original system of equations. Note that A and B are modified on exit if `equed ≠ 'N'`, and the solution to the equilibrated system is:

`inv(diag(s)) * X.`

`a`

If `fact = 'E'` and `equed = 'Y'`, overwritten by `diag(s) * A * diag(s)`.

`af`

If `fact = 'N'`, `af` is an output argument and on exit returns the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U * D * U^T$ or $A = L * D * L^T$.

`b`

If `equed = 'N'`, B is not modified.

If `equed = 'Y'`, B is overwritten by `diag(s) * B`.

`s`

This array is an output argument if `fact ≠ 'F'`. Each element of this array is a power of the radix. See the description of `s` in *Input Arguments* section.

`rcond`

REAL for `chesvxx`

DOUBLE PRECISION for `zhesvxx`.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If `rcond` is less than the machine precision, in particular, if `rcond = 0`, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

`rpvgrw`

REAL for `chesvxx`

DOUBLE PRECISION for `zhesvxx`.

Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the *LU* factorization of the (equilibrated) matrix *A* could be poor. This also means that the solution *X*, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < \text{info} \leq n$, this parameter contains the reciprocal pivot growth factor for the leading *info* columns of *A*.

berr

REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of *A* or *B* that makes $x(j)$ an exact solution.

err_bnds_norm

REAL for chesvxx

DOUBLE PRECISION for zhesvxx.

Array of size *nrhs* by *n_err_bnds*. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt(n) * \text{slamch}(\epsilon)$ for chesvxx and $\sqrt(n) * \text{dlamch}(\epsilon)$ for zhesvxx.

err=2

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt(n) * \text{slamch}(\epsilon)$ for chesvxx and $\sqrt(n) * \text{dlamch}(\epsilon)$ for zhesvxx. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt(n) * \text{slamch}(\epsilon)$ for chesvxx and

$\text{sqrt}(n) * \text{diamch}(\epsilon)$ for `zhesvxx` to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s^* a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

`err_bnds_comp`

REAL for `chesvxx`
DOUBLE PRECISION for `zhesvxx`.

Array of size $nrhs$ by n_err_bnds . For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params(3) = 0.0`), then `err_bnds_comp` is not accessed. If $n_err_bnds < 3$, then at most the first $(:, n_err_bnds)$ entries are returned.

The first index in `err_bnds_comp(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:, err)` contains the following three fields:

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\text{sqrt}(n) * \text{slamch}(\epsilon)$ for <code>chesvxx</code> and $\text{sqrt}(n) * \text{diamch}(\epsilon)$ for <code>zhesvxx</code> .
<code>err=2</code>	"Guaranteed" error bbound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\text{sqrt}(n) * \text{slamch}(\epsilon)$ for <code>chesvxx</code> and $\text{sqrt}(n) * \text{diamch}(\epsilon)$ for <code>zhesvxx</code> . This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\text{sqrt}(n) * \text{slamch}(\epsilon)$ for <code>chesvxx</code> and $\text{sqrt}(n) * \text{diamch}(\epsilon)$ for <code>zhesvxx</code> to determine

if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

$$\|Z\|_{\infty} \cdot \|Z^{-1}\|_{\infty}$$

Let $z = s^*(a^*\text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a^*\text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

ipiv If $\text{fact} = 'N'$, *ipiv* is an output argument and on exit contains details of the interchanges and the block structure D , as determined by [ssytrf](#) for single precision flavors and [dsytrf](#) for double precision flavors.

equed If $\text{fact} \neq 'F'$, then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

params If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

info INTEGER. If $\text{info} = 0$, the execution is successful. The solution to every right-hand side is guaranteed.

If $\text{info} = -i$, the i -th parameter had an illegal value.

If $0 < \text{info} \leq n$: $U(\text{info}, \text{info})$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $\text{info} = n+j$: The solution corresponding to the j -th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested *params(3) = 0.0*, then the j -th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that *err_bnds_norm(j, 1) = 0.0* or *err_bnds_comp(j, 1) = 0.0*). See the definition of *err_bnds_norm* and *err_bnds_comp* for *err = 1*. To get information about all of the right-hand sides, check *err_bnds_norm* or *err_bnds_comp*.

See Also

[Matrix Storage Schemes](#)

?spsv

Computes the solution to the system of linear equations with a real or complex symmetric matrix A stored in packed format, and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call sspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call dspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

```
call cspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zspsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call spsv( ap, b [,uplo] [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>spsv( int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, <datatype>* ap, lapack_int* ipiv, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the real or complex system of linear equations $A*X = B$, where A is an n -by- n symmetric matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U*D*U^T$ or $A = L*D*L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A*X = B$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored:
	If <i>uplo</i> = 'U', the upper triangle of A is stored.
	If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>ap, b</i>	REAL for sspsv DOUBLE PRECISION for dsspsv COMPLEX for cspsv DOUBLE COMPLEX for zspsv. Arrays: <i>ap</i> (*), <i>b</i> (<i>ldb</i> ,*).
	The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The array <i>ap</i> contains the factor U or L , as specified by <i>uplo</i> , in <i>packed storage</i> (see Matrix Storage Schemes).

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

ldb

INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

ap

The block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by [?sptrf](#), stored as a packed triangular matrix in the same storage format as A .

b

If $info = 0$, b is overwritten by the solution matrix X .

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of D , as determined by [?sptrf](#).

If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the i -th row and column of A was interchanged with the k -th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spsv` interface are as follows:

ap

Holds the array A of size $(n * (n+1) / 2)$.

b

Holds the matrix B of size $(n, nrhs)$.

ipiv

Holds the vector with the number of elements n .

uplo

Must be '`U`' or '`L`'. The default value is '`U`'.

?spsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric matrix A stored in packed format, and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call sspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, iwork, info )

call dsspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, iwork, info )

call cspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )

call zspsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )
```

Fortran 95:

```
call spsvx( ap, b, x [,uplo] [,afp] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

C:

```
lapack_int LAPACKE_sspsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const float* ap, float* afp, lapack_int* ipiv, const float* b,
lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_dsspsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const double* ap, double* afp, lapack_int* ipiv, const double* b,
lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

lapack_int LAPACKE_cspsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_float* ap, lapack_complex_float* afp, lapack_int* ipiv,
const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zspsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double* ap, lapack_complex_double* afp,
lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- **Fortran:** mkl.fi
- **Fortran 95:** lapack.f90
- **C:** mkl.h

Description

The routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations $A \cdot X = B$, where A is a n -by- n symmetric matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?spsvx performs the following steps:

1. If $\text{fact} = \text{'N'}$, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U \cdot D \cdot U^T$ or $A = L \cdot D \cdot L^T$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>fact</i>	CHARACTER*1. Must be 'F' or 'N'.
	Specifies whether or not the factored form of the matrix A has been supplied on entry.
	If $fact = 'F'$: on entry, afp and $ipiv$ contain the factored form of A . Arrays ap , afp , and $ipiv$ are not modified.
	If $fact = 'N'$, the matrix A is copied to afp and factored.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored and how A is factored:
	If $uplo = 'U'$, the array ap stores the upper triangular part of the symmetric matrix A , and A is factored as $U*D*U^T$.
	If $uplo = 'L'$, the array ap stores the lower triangular part of the symmetric matrix A ; A is factored as $L*D*L^T$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides, the number of columns in B ; $nrhs \geq 0$.
<i>ap, afp, b, work</i>	REAL for sspsvx DOUBLE PRECISION for dspsvx COMPLEX for cspsvx DOUBLE COMPLEX for zspsvx. Arrays: $ap(*)$, $afp(*)$, $b(ldb,*)$, $work(*)$.
	The array ap contains the upper or lower triangle of the symmetric matrix A in <i>packed storage</i> (see Matrix Storage Schemes).
	The array afp is an input argument if $fact = 'F'$. It contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U*D*U^T$ or $A = L*D*L^T$ as computed by ?sptrf , in the same storage format as A .
	The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.
	$work(*)$ is a workspace array.

The dimension of arrays *ap* and *afp* must be at least $\max(1, n(n+1)/2)$; the second dimension of *b* must be at least $\max(1, nrhs)$; the dimension of *work* must be at least $\max(1, 3*n)$ for real flavors and $\max(1, 2*n)$ for complex flavors.

ldb

INTEGER. The leading dimension of *b*; $ldb \geq \max(1, n)$.

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array *ipiv* is an input argument if *fact* = 'F'. It contains details of the interchanges and the block structure of *D*, as determined by [?sptrf](#).

If *ipiv*(*i*) = *k* > 0, then d_{ii} is a 1-by-1 diagonal block, and the *i*-th row and column of *A* was interchanged with the *k*-th row and column.

If *uplo* = 'U' and *ipiv*(*i*) = *ipiv*(*i*-1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*-1, and (*i*-1)-th row and column of *A* was interchanged with the *m*-th row and column.

If *uplo* = 'L' and *ipiv*(*i*) = *ipiv*(*i*+1) = -*m* < 0, then *D* has a 2-by-2 block in rows/columns *i* and *i*+1, and (*i*+1)-th row and column of *A* was interchanged with the *m*-th row and column.

ldx

INTEGER. The leading dimension of the output array *x*; $ldx \geq \max(1, n)$.

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, n)$; used in real flavors only.

rwork

REAL for `cspsvx`

DOUBLE PRECISION for `zspsvx`.

Workspace array, DIMENSION at least $\max(1, n)$; used in complex flavors only.

Output Parameters

x

REAL for `sspsvx`

DOUBLE PRECISION for `dspsvx`

COMPLEX for `cspsvx`

DOUBLE COMPLEX for `zspsvx`.

Array, DIMENSION (*ldx*, *).

If *info* = 0 or *info* = *n*+1, the array *x* contains the solution matrix *X* to the system of equations. The second dimension of *x* must be at least $\max(1, nrhs)$.

afp, *ipiv*

These arrays are output arguments if *fact* = 'N'. See the description of *afp*, *ipiv* in *Input Arguments* section.

rcond

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix A . If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr, berr

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION at least $\max(1, nrhs)$. Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

*info*INTEGER. If $info = 0$, the execution is successful.If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n + 1$, then D is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spsvx` interface are as follows:

*ap*Holds the array A of size $(n * (n+1) / 2)$.*b*Holds the matrix B of size $(n, nrhs)$.*x*Holds the matrix X of size $(n, nrhs)$.*afp*Holds the array AF of size $(n * (n+1) / 2)$.*ipiv*Holds the vector with the number of elements n .*ferr*Holds the vector with the number of elements $nrhs$.*berr*Holds the vector with the number of elements $nrhs$.*uplo*Must be '`U`' or '`L`'. The default value is '`U`'.*fact*

Must be '`N`' or '`F`'. The default value is '`N`'. If $fact = 'F'$, then both arguments af and $ipiv$ must be present; otherwise, an error is returned.

?hpsv

Computes the solution to the system of linear equations with a Hermitian matrix A stored in packed format, and multiple right-hand sides.

Syntax

FORTRAN 77:

```
call chpsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
call zhpsv( uplo, n, nrhs, ap, ipiv, b, ldb, info )
```

Fortran 95:

```
call hpsv( ap, b [,uplo] [,ipiv] [,info] )
```

C:

```
lapack_int LAPACKE_<?>hpsv( int matrix_layout, char uplo, lapack_int n, lapack_int
nrhs, <datatype>* ap, lapack_int* ipiv, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves for X the system of linear equations $A^*X = B$, where A is an n -by- n Hermitian matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as $A = U*D*U^H$ or $A = L*D*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations $A^*X = B$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular part of A is stored:
	If <i>uplo</i> = 'U', the upper triangle of A is stored.
	If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in B ; $nrhs \geq 0$.
<i>ap</i> , <i>b</i>	COMPLEX for chpsv DOUBLE COMPLEX for zhpsv.

Arrays: $ap(*)$, $b(ldb,*)$.

The dimension of ap must be at least $\max(1, n(n+1)/2)$. The array ap contains the factor U or L , as specified by $uplo$, in *packed storage* (see [Matrix Storage Schemes](#)).

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

ldb

INTEGER. The leading dimension of b ; $ldb \geq \max(1, n)$.

Output Parameters

ap

The block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by [?hptrf](#), stored as a packed triangular matrix in the same storage format as A .

b

If $info = 0$, b is overwritten by the solution matrix X .

$ipiv$

INTEGER.
Array, DIMENSION at least $\max(1, n)$. Contains details of the interchanges and the block structure of D , as determined by [?hptrf](#).

If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 block, and the i -th row and column of A was interchanged with the k -th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

$info$

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpsv` interface are as follows:

ap

Holds the array A of size $(n*(n+1)/2)$.

b

Holds the matrix B of size $(n, nrhs)$.

$ipiv$

Holds the vector with the number of elements n .

$uplo$

Must be '`U`' or '`L`'. The default value is '`U`'.

?hpsvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a Hermitian matrix A stored in packed format, and provides error bounds on the solution.

Syntax

FORTRAN 77:

```
call chpsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )
```

```
call zhpsvx( fact, uplo, n, nrhs, ap, afp, ipiv, b, ldb, x, ldx, rcond, ferr, berr,
work, rwork, info )
```

Fortran 95:

```
call hpsvx( ap, b, x [,uplo] [,afp] [,ipiv] [,fact] [,ferr] [,berr] [,rcond] [,info] )
```

C:

```
lapack_int LAPACKE_chpsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_float* ap, lapack_complex_float* afp, lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* rcond, float* ferr, float* berr );  

lapack_int LAPACKE_zhpsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double* ap, lapack_complex_double* afp,
lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $A^*X = B$, where A is a n -by- n Hermitian matrix stored in packed format, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?hpsvx performs the following steps:

1. If $fact = 'N'$, the diagonal pivoting method is used to factor the matrix A . The form of the factorization is $A = U^*D^*U^H$ or $A = L^*D^*L^H$, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some $d_{i,i} = 0$, so that D is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for X and compute error bounds as described below.
3. The system of equations is solved for X using the factored form of A .
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

fact

CHARACTER*1. Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix A has been supplied on entry.

If $\text{fact} = \text{'F'}$: on entry, afp and ipiv contain the factored form of A . Arrays ap , afp , and ipiv are not modified.

If $\text{fact} = \text{'N'}$, the matrix A is copied to afp and factored.

uplo

CHARACTER*1. Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If $\text{uplo} = \text{'U'}$, the array ap stores the upper triangular part of the Hermitian matrix A , and A is factored as U^*D*U^H .

If $\text{uplo} = \text{'L'}$, the array ap stores the lower triangular part of the Hermitian matrix A , and A is factored as L^*D*L^H .

*n*INTEGER. The order of matrix A ; $n \geq 0$.*nrhs*INTEGER. The number of right-hand sides, the number of columns in B ; $\text{nrhs} \geq 0$.*ap, afp, b, work*

COMPLEX for chpsvx

DOUBLE COMPLEX for zhpsvx.

Arrays: $\text{ap}(*)$, $\text{afp}(*)$, $\text{b}(1:\text{ldb}, *)$, $\text{work}(*)$.

The array ap contains the upper or lower triangle of the Hermitian matrix A in *packed storage* (see [Matrix Storage Schemes](#)).

The array afp is an input argument if $\text{fact} = \text{'F'}$. It contains the block diagonal matrix D and the multipliers used to obtain the factor U or L from the factorization $A = U^*D*U^H$ or $A = L^*D*L^H$ as computed by [?hptrf](#), in the same storage format as A .

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

$\text{work}(*)$ is a workspace array.

The dimension of arrays ap and afp must be at least $\max(1, n(n+1)/2)$; the second dimension of b must be at least $\max(1, \text{nrhs})$; the dimension of work must be at least $\max(1, 2*n)$.

*ldb*INTEGER. The leading dimension of b ; $\text{ldb} \geq \max(1, n)$.*ipiv*

INTEGER.

Array, DIMENSION at least $\max(1, n)$. The array ipiv is an input argument if $\text{fact} = \text{'F'}$. It contains details of the interchanges and the block structure of D , as determined by [?hptrf](#).

If $ipiv(i) = k > 0$, then d_{ii} is a 1-by-1 diagonal block, and the i -th row and column of A was interchanged with the k -th row and column.

If $uplo = 'U'$ and $ipiv(i) = ipiv(i-1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i-1$, and $(i-1)$ -th row and column of A was interchanged with the m -th row and column.

If $uplo = 'L'$ and $ipiv(i) = ipiv(i+1) = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and $(i+1)$ -th row and column of A was interchanged with the m -th row and column.

ldx

INTEGER. The leading dimension of the output array x ; $ldx \geq \max(1, n)$.

rwork

REAL for chpsvx

DOUBLE PRECISION for zhpsvx.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

x

COMPLEX for chpsvx

DOUBLE COMPLEX for zhpsvx.

Array, DIMENSION ($ldx, *$).

If $info = 0$ or $info = n+1$, the array x contains the solution matrix X to the system of equations. The second dimension of x must be at least $\max(1, nrhs)$.

afp, ipiv

These arrays are output arguments if $fact = 'N'$. See the description of *afp, ipiv* in *Input Arguments* section.

rcond

REAL for chpsvx

DOUBLE PRECISION for zhpsvx.

An estimate of the reciprocal condition number of the matrix A . If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr

REAL for chpsvx

DOUBLE PRECISION for zhpsvx.

Array, DIMENSION at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x(j)$ (the j -th column of the solution matrix X). If $xtrue$ is the true solution corresponding to $x(j)$, $ferr(j)$ is an estimated upper bound for the magnitude of the largest element in $(x(j) - xtrue)$ divided by the magnitude of the largest element in $x(j)$. The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

berr

REAL for chpsvx

DOUBLE PRECISION for zhpsvx.

Array, DIMENSION at least $\max(1, \text{nrhs})$. Contains the component-wise relative backward error for each solution vector $x(j)$, that is, the smallest relative change in any element of A or B that makes $x(j)$ an exact solution.

info

INTEGER. If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and $i \leq n$, then d_{ii} is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution and error bounds could not be computed; $rcond = 0$ is returned.

If $info = i$, and $i = n + 1$, then D is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpsvx` interface are as follows:

ap

Holds the array A of size $(n * (n+1) / 2)$.

b

Holds the matrix B of size (n, nrhs) .

x

Holds the matrix X of size (n, nrhs) .

afp

Holds the array AF of size $(n * (n+1) / 2)$.

ipiv

Holds the vector with the number of elements n .

ferr

Holds the vector with the number of elements nrhs .

berr

Holds the vector with the number of elements nrhs .

uplo

Must be '`U`' or '`L`'. The default value is '`U`'.

fact

Must be '`N`' or '`F`'. The default value is '`N`'. If $fact = 'F'$, then both arguments af and $ipiv$ must be present; otherwise, an error is returned.

Least Square and Eigenvalue Problem Routines

This section includes descriptions of LAPACK [computational routines](#) and [driver routines](#) for solving linear least squares problems, eigenvalue and singular value problems, and performing a number of related computational tasks. For a full reference on LAPACK routines and related information see [\[LUG\]](#).

Least Squares Problems. A typical *least squares problem* is as follows: given a matrix A and a vector b , find the vector x that minimizes the sum of squares $\sum_i ((Ax)_i - b_i)^2$ or, equivalently, find the vector x that minimizes the 2-norm $\|Ax - b\|_2$.

In the most usual case, A is an m -by- n matrix with $m \geq n$ and $\text{rank}(A) = n$. This problem is also referred to as finding the *least squares solution* to an *overdetermined* system of linear equations (here we have more equations than unknowns). To solve this problem, you can use the QR factorization of the matrix A (see [QR Factorization](#)).

If $m < n$ and $\text{rank}(A) = m$, there exist an infinite number of solutions x which exactly satisfy $Ax = b$, and thus minimize the norm $\|Ax - b\|_2$. In this case it is often useful to find the unique solution that minimizes $\|x\|_2$. This problem is referred to as finding the *minimum-norm solution* to an *underdetermined* system of linear equations (here we have more unknowns than equations). To solve this problem, you can use the LQ factorization of the matrix A (see [LQ Factorization](#)).

In the general case you may have a *rank-deficient least squares problem*, with $\text{rank}(A) < \min(m, n)$: find the *minimum-norm least squares solution* that minimizes both $\|x\|_2$ and $\|Ax - b\|^2$. In this case (or when the rank of A is in doubt) you can use the QR factorization with pivoting or *singular value decomposition* (see [Singular Value Decomposition](#)).

Eigenvalue Problems. The eigenvalue problems (from German *eigen* "own") are stated as follows: given a matrix A , find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation

$$Az = \lambda z \text{ (right eigenvectors } z\text{)}$$

or the equation

$$z^H A = \lambda z^H \text{ (left eigenvectors } z\text{).}$$

If A is a real symmetric or complex Hermitian matrix, the above two equations are equivalent, and the problem is called a *symmetric* eigenvalue problem. Routines for solving this type of problems are described in the section [Symmetric Eigenvalue Problems](#).

Routines for solving eigenvalue problems with nonsymmetric or non-Hermitian matrices are described in the section [Nonsymmetric Eigenvalue Problems](#).

The library also includes routines that handle *generalized symmetric-definite eigenvalue problems*: find the eigenvalues λ and the corresponding eigenvectors x that satisfy one of the following equations:

$$Az = \lambda Bz, ABz = \lambda z, \text{ or } BAz = \lambda z,$$

where A is symmetric or Hermitian, and B is symmetric positive-definite or Hermitian positive-definite. Routines for reducing these problems to standard symmetric eigenvalue problems are described in the section [Generalized Symmetric-Definite Eigenvalue Problems](#).

To solve a particular problem, you usually call several computational routines. Sometimes you need to combine the routines of this chapter with other LAPACK routines described in "LAPACK Routines: Linear Equations" as well as with BLAS routines described in "BLAS and Sparse BLAS Routines".

For example, to solve a set of least squares problems minimizing $\|Ax - b\|^2$ for all columns b of a given matrix B (where A and B are real matrices), you can call [?geqr](#) to form the factorization $A = QR$, then call [?ormqr](#) to compute $C = Q^H B$ and finally call the BLAS routine [?trsm](#) to solve for X the system of equations $RX = C$.

Another way is to call an appropriate driver routine that performs several tasks in one call. For example, to solve the least squares problem the driver routine [?gels](#) can be used.

Computational Routines

In the sections that follow, the descriptions of LAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

[Orthogonal Factorizations](#)

[Singular Value Decomposition](#)

[Symmetric Eigenvalue Problems](#)

[Generalized Symmetric-Definite Eigenvalue Problems](#)

[Nonsymmetric Eigenvalue Problems](#)

[Generalized Nonsymmetric Eigenvalue Problems](#)

[Generalized Singular Value Decomposition](#)

See also the respective [driver routines](#).

Orthogonal Factorizations

This section describes the LAPACK routines for the QR (RQ) and LQ (QL) factorization of matrices. Routines for the RZ factorization as well as for generalized QR and RQ factorizations are also included.

QR Factorization. Assume that A is an m -by- n matrix to be factored.

If $m \geq n$, the QR factorization is given by

where R is an n -by- n upper triangular matrix with real diagonal elements, and Q is an m -by- m orthogonal (or unitary) matrix.

You can use the QR factorization for solving the following least squares problem: minimize $\|Ax - b\|^2$ where A is a full-rank m -by- n matrix ($m \geq n$). After factoring the matrix, compute the solution x by solving $Rx = (Q_1)^T b$.

If $m < n$, the QR factorization is given by

$$A = QR = Q(R_1 R_2)$$

where R is trapezoidal, R_1 is upper triangular and R_2 is rectangular.

The LAPACK routines do not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation.

LQ Factorization LQ factorization of an m -by- n matrix A is as follows. If $m \leq n$,

$$A = (L, 0) \mathcal{Q} = (L, 0) \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = (LQ_1)$$

where L is an m -by- m lower triangular matrix with real diagonal elements, and Q is an n -by- n orthogonal (or unitary) matrix.

If $m > n$, the LQ factorization is

11111

where L_1 is an n -by- n lower triangular matrix, L_2 is rectangular, and Q is an n -by- n orthogonal (or unitary) matrix.

You can use the LQ factorization to find the minimum-norm solution of an underdetermined system of linear equations $Ax = b$ where A is an m -by- n matrix of rank m ($m < n$). After factoring the matrix, compute the solution vector x as follows: solve $Ly = b$ for y , and then compute $x = (Q_1)^H y$.

[Table "Computational Routines for Orthogonal Factorization"](#) lists LAPACK routines (FORTRAN 77 interface) that perform orthogonal factorization of matrices. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Orthogonal Factorization

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	geqrf geqrfp	geqpf geqp3	orgqr ungqr	ormqr unmqr
general matrices, blocked QR factorization	?geqrt			?gemqrt
general matrices, RQ factorization	gerqf		orgrq ungrq	ormrq unmrq
general matrices, LQ factorization	gelqf		orglq unqlq	ormlq unmlq
general matrices, QL factorization	geqlf		orgql ungql	ormql unmql
trapezoidal matrices, RZ factorization	tzrzf			ormrz unmrz
pair of matrices, generalized QR factorization	ggqrf			
pair of matrices, generalized RQ factorization	ggrqf			
triangular-pentagonal matrices, blocked QR factorization	?tpqrt			?tpmqrt

?geqrf

Computes the QR factorization of a general m -by- n matrix.

Syntax

FORTRAN 77:

```
call sgeqrf(m, n, a, lda, tau, work, lwork, info)
call dgeqrf(m, n, a, lda, tau, work, lwork, info)
call cgeqrf(m, n, a, lda, tau, work, lwork, info)
call zgeqrf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call gqrf(a [, tau] [,info])
```

C:

```
lapack_int LAPACKE_<?>geqrf( int matrix_layout, lapack_int m, lapack_int n, <datatype>* a, lapack_int lda, <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the QR factorization of a general m -by- n matrix A (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in A ($n \geq 0$).
<i>a</i> , <i>work</i>	REAL for sgeqr DOUBLE PRECISION for dgeqr COMPLEX for cgeqr DOUBLE COMPLEX for zgeqr. Arrays: <i>a</i> (<i>lda</i> ,*) contains the matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array (<i>lwork</i> $\geq n$). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See Application Notes for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: If $m \geq n$, the elements below the diagonal are overwritten by the details of the unitary matrix Q , and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix R . If $m < n$, the strictly lower triangular part is overwritten by the details of the unitary matrix Q , and the remaining elements are overwritten by the corresponding elements of the m -by- n upper trapezoidal matrix R .
<i>tau</i>	REAL for sgeqr DOUBLE PRECISION for dgeqr

COMPLEX for `cgeqrf`

DOUBLE COMPLEX for `zgeqrf`.

Array, DIMENSION at least $\max(1, \min(m, n))$. Contains additional information on the matrix Q .

`work(1)` If $info = 0$, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info` INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geqrf` interface are the following:

`a` Holds the matrix A of size (m, n) .

`tau` Holds the vector of length $\min(m, n)$

Application Notes

For better performance, try using $lwork = n * blocksize$, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$(4/3)n^3 \quad \text{if } m = n,$$

$$(2/3)n^2(3m-n) \quad \text{if } m > n,$$

$$(2/3)m^2(3n-m) \quad \text{if } m < n.$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

?geqrf (this routine)	to factorize $A = QR$;
ormqr	to compute $C = Q^T B$ (for real matrices);
unmqr	to compute $C = Q^H B$ (for complex matrices);
trsm (a BLAS routine)	to solve $R^T X = C$.

(The columns of the computed X are the least squares solution vectors x .)

To compute the elements of Q explicitly, call

orgqr	(for real matrices)
ungqr	(for complex matrices).

See Also

[mkl_progress](#)

?geqrfp

Computes the QR factorization of a general m -by- n matrix with non-negative diagonal elements.

Syntax

FORTRAN 77:

```
call sgeqrfp(m, n, a, lda, tau, work, lwork, info)
call dgeqrfp(m, n, a, lda, tau, work, lwork, info)
call cgeqrfp(m, n, a, lda, tau, work, lwork, info)
call zgeqrfp(m, n, a, lda, tau, work, lwork, info)
```

C:

```
lapack_int LAPACKE_<?>geqrfp( int matrix_layout, lapack_int m, lapack_int n,
<datatype>* a, lapack_int lda, <datatype>* tau );
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine forms the QR factorization of a general m -by- n matrix A (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in A ($n \geq 0$).
<i>a, work</i>	REAL for sgeqrfp DOUBLE PRECISION for dgeqrfp COMPLEX for cgeqrfp DOUBLE COMPLEX for zgeqrfp. Arrays: <i>a</i> (<i>lda</i> ,*) contains the matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array (<i>lwork</i> $\geq n$). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See Application Notes for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: If $m \geq n$, the elements below the diagonal are overwritten by the details of the unitary matrix Q , and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix R . If $m < n$, the strictly lower triangular part is overwritten by the details of the unitary matrix Q , and the remaining elements are overwritten by the corresponding elements of the m -by- n upper trapezoidal matrix R . The diagonal elements of the matrix R are non-negative.
<i>tau</i>	REAL for sgeqrfp DOUBLE PRECISION for dgeqrfp COMPLEX for cgeqrfp DOUBLE COMPLEX for zgeqrfp. Array, DIMENSION at least $\max(1, \min(m, n))$. Contains additional information on the matrix Q .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geqrfp` interface are the following:

`a` Holds the matrix A of size (m,n) .

`tau` Holds the vector of length $\min(m,n)$

Application Notes

For better performance, try using $\text{lwork} = n * \text{blocksize}$, where blocksize is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of lwork for the first run or set $\text{lwork} = -1$.

If you choose the first option and set any of admissible lwork sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set $\text{lwork} = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set lwork to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$(4/3)n^3 \quad \text{if } m = n,$$

$$(2/3)n^2(3m-n) \quad \text{if } m > n,$$

$$(2/3)m^2(3n-m) \quad \text{if } m < n.$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

`?geqrfp` (this routine) to factorize $A = QR$;

`ormqr` to compute $C = Q^T B$ (for real matrices);

`unmqr` to compute $C = Q^H B$ (for complex matrices);

`trsm` (a BLAS routine) to solve $R^*X = C$.

(The columns of the computed X are the least squares solution vectors x .)

To compute the elements of Q explicitly, call

[orgqr](#) (for real matrices)
[ungqr](#) (for complex matrices).

See Also

[mkl_progress](#)

?geqrt

Computes a blocked QR factorization of a general real or complex matrix using the compact WY representation of Q.

Syntax

FORTRAN 77:

```
call sgeqrt(m, n, nb, a, lda, t, ldt, work, info)
call dgeqrt(m, n, nb, a, lda, t, ldt, work, info)
call cgeqrt(m, n, nb, a, lda, t, ldt, work, info)
call zgeqrt(m, n, nb, a, lda, t, ldt, work, info)
```

Fortran 95:

```
call geqrt(a, t, nb [, info])
```

C:

```
lapack_int LAPACKE_<?>geqrt( int matrix_layout, lapack_int m, lapack_int n, lapack_int nb, <datatype>* a, lapack_int lda, <datatype>* t, lapack_int ldt );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The strictly lower triangular matrix V contains the elementary reflectors $H(i)$ in the i th column below the diagonal. For example, if $m=5$ and $n=3$, the matrix V is

$$V = \begin{bmatrix} 1 & & \\ v_1 & 1 & \\ v_1 & v_2 & 1 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

where v_i represents one of the vectors that define $H(i)$. The vectors are returned in the lower triangular part of array `a`.

NOTE

The 1s along the diagonal of V are not stored in a .

Let $k = \min(m, n)$. The number of blocks is $b = \text{ceiling}(k/nb)$, where each block is of order nb except for the last block, which is of order $ib = k - (b-1)*nb$. For each of the b blocks, a upper triangular block reflector factor is computed: t_1, t_2, \dots, t_b . The nb -by- nb (and ib -by- ib for the last block) ts are stored in the nb -by- n array t as

$$t = (t_1 \ t_2 \ \dots \ t_b).$$

Input Parameters

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
nb	INTEGER. The block size to be used in the blocked QR ($\min(m, n) \geq nb \geq 1$).
$a, work$	REAL for sgeqrt DOUBLE PRECISION for dgeqrt COMPLEX for cgeqrt COMPLEX*16 for zgeqrt. Arrays: a DIMENSION (lda, n) contains the m -by- n matrix A . $work$ DIMENSION (nb, n) is a workspace array.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
ldt	INTEGER. The leading dimension of t ; at least nb .

Output Parameters

a	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the $\min(m, n)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$). The elements below the diagonal are the columns of V .
t	REAL for sgeqrt DOUBLE PRECISION for dgeqrt COMPLEX for cgeqrt COMPLEX*16 for zgeqrt. Array, DIMENSION ($ldt, \min(m, n)$). The upper triangular block reflectors stored in compact form as a sequence of upper triangular blocks.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info < 0$ and $info = -i$, the i th argument had an illegal value.

?gemqrt

Multiples a general matrix by the orthogonal/unitary matrix Q of the QR factorization formed by ?geqrt.

Syntax**FORTRAN 77:**

```
call sgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
call dgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
call cgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
call zgemqrt(side, trans, m, n, k, nb, v, ldv, t, ldt, c, ldc, work, info)
```

Fortran 95:

```
call gemqrt( v, t, c, k, nb [, trans] [, side] [, info])
```

C:

```
lapack_int LAPACKE_<?>gemqrt(int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int nb, const <datatype>* v, lapack_int ldv,
const <datatype>* t, lapack_int ldt, <datatype>* c, lapack_int ldc)
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The ?gemqrt routine overwrites the general real or complex m -by- n matrix C with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	Q^*C	C^*Q
<i>trans</i> = 'T':	Q^T*C	C^*Q^T
<i>trans</i> = 'C':	Q^H*C	C^*Q^H

where Q is a complex orthogonal distributed matrix defined as the product of k elementary reflectors

$Q = H(1) H(2) \dots H(k) = I - V^* T^* V^T$ for real flavors, and

$Q = H(1) H(2) \dots H(k) = I - V^* T^* V^H$ for complex flavors,

generated using the compact WY representation as returned by [geqrt](#). Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) CHARACTER
= 'L':	apply Q , Q^T , or Q^H from the left.
= 'R':	apply Q , Q^T , or Q^H from the right.

<i>trans</i>	(global) CHARACTER
= 'N',	no transpose, apply Q .
= 'T',	transpose, apply Q^T .
= 'C',	transpose, apply Q^H .

m (global) INTEGER. The number of rows in the matrix C , ($m \geq 0$).

n (global) INTEGER. The number of columns in the matrix C , ($n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints:
 If $side = 'L'$, $m \geq k \geq 0$
 If $side = 'R'$, $n \geq k \geq 0$.

nb INTEGER.
 The block size used for the storage of t , $k \geq nb \geq 1$. This must be the same value of *nb* used to generate t in [geqrt](#).

v REAL for sgemqrt
 DOUBLE PRECISION for dgemqrt
 COMPLEX for cgemqrt
 COMPLEX*16 for zgemqrt.
 Array, DIMENSION (ldv, k).
 The i th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by [geqrt](#) in the first k columns of its array argument *a*.

ldv INTEGER. The leading dimension of the array *v*.
 if $side = 'L'$, *ldv* must be at least $\max(1, m)$;
 if $side = 'R'$, *ldv* must be at least $\max(1, n)$.

t REAL for sgemqrt
 DOUBLE PRECISION for dgemqrt
 COMPLEX for cgemqrt
 COMPLEX*16 for zgemqrt.
 Array, DIMENSION (ldt, k).
 The upper triangular factors of the block reflectors as returned by [geqrt](#), stored as an nb -by- n matrix.

ldt INTEGER. The leading dimension of the array *t*. *ldt* must be at least *nb*.

c REAL for sgemqrt
 DOUBLE PRECISION for dgemqrt
 COMPLEX for cgemqrt
 COMPLEX*16 for zgemqrt.
 The m -by- matrix C .

ldc INTEGER. The leading dimension of the array *c*. *ldc* must be at least $\max(1, m)$.

work REAL for sgemqrt

DOUBLE PRECISION **for** dgemqrt
 COMPLEX **for** cgemqrt
 COMPLEX*16 **for** zgemqrt.
Workspace array.
If *side* = 'L' DIMENSION *n*nb*.
If *side* = 'R' DIMENSION *m*nb*.

Output Parameters

c Overwritten by the product Q^*C , C^*Q , Q^T*C , C^*Q^T , Q^H*C , or C^*Q^H .
info (global) INTEGER.
 = 0: the execution is successful.
 < 0: if *info* = -*i*, the *i*th argument had an illegal value.

?geqpf

Computes the QR factorization of a general *m*-by-*n* matrix with pivoting.

Syntax

FORTRAN 77:

```
call sgeqpf(m, n, a, lda, jpvt, tau, work, info)
call dgeqpf(m, n, a, lda, jpvt, tau, work, info)
call cgeqpf(m, n, a, lda, jpvt, tau, work, rwork, info)
call zgeqpf(m, n, a, lda, jpvt, tau, work, rwork, info)
```

Fortran 95:

```
call geqpf(a, jpvt [, tau] [, info])
```

C:

```
lapack_int LAPACKE_<?>geqpf( int matrix_layout, lapack_int m, lapack_int n, <datatype>* a, lapack_int lda, lapack_int* jpvt, <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine is deprecated and has been replaced by routine [geqp3](#).

The routine ?geqpf forms the QR factorization of a general *m*-by-*n* matrix *A* with column pivoting: $A^*P = Q^*R$ (see [Orthogonal Factorizations](#)). Here *P* denotes an *n*-by-*n* permutation matrix.

The routine does not form the matrix *Q* explicitly. Instead, *Q* is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with *Q* in this representation.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in A ($n \geq 0$).
<i>a, work</i>	REAL for sgeqpf DOUBLE PRECISION for dgeqpf COMPLEX for cgeqpf DOUBLE COMPLEX for zgeqpf. Arrays: <i>a</i> (<i>lda,*</i>) contains the matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> (<i>lwork</i>) is a workspace array. The size of the <i>work</i> array must be at least $\max(1, 3*n)$ for real flavors and at least $\max(1, n)$ for complex flavors.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>jpvt</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. On entry, if $jpvt(i) > 0$, the <i>i</i> -th column of A is moved to the beginning of $A*P$ before the computation, and fixed in place during the computation. If $jpvt(i) = 0$, the <i>i</i> th column of A is a free column (that is, it may be interchanged during the computation with any other free column).
<i>rwork</i>	REAL for cgeqpf DOUBLE PRECISION for zgeqpf. A workspace array, DIMENSION at least $\max(1, 2*n)$.

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: If $m \geq n$, the elements below the diagonal are overwritten by the details of the unitary (orthogonal) matrix Q , and the upper triangle is overwritten by the corresponding elements of the upper triangular matrix R . If $m < n$, the strictly lower triangular part is overwritten by the details of the matrix Q , and the remaining elements are overwritten by the corresponding elements of the m -by- n upper trapezoidal matrix R .
<i>tau</i>	REAL for sgeqpf DOUBLE PRECISION for dgeqpf COMPLEX for cgeqpf DOUBLE COMPLEX for zgeqpf. Array, DIMENSION at least $\max(1, \min(m, n))$. Contains additional information on the matrix Q .

<i>jpvt</i>	Overwritten by details of the permutation matrix P in the factorization $A^*P = Q^*R$. More precisely, the columns of A^*P are the columns of A in the following order: $jpvt(1), jpvt(2), \dots, jpvt(n)$.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geqpf` interface are the following:

<i>a</i>	Holds the matrix A of size (m,n) .
<i>jpvt</i>	Holds the vector of length n .
<i>tau</i>	Holds the vector of length $\min(m,n)$

Application Notes

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\epsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$\begin{aligned} (4/3)n^3 & \quad \text{if } m = n, \\ (2/3)n^2(3m-n) & \quad \text{if } m > n, \\ (2/3)m^2(3n-m) & \quad \text{if } m < n. \end{aligned}$$

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

<code>?geqpf</code> (this routine)	to factorize $A^*P = Q^*R$;
<code>ormqr</code>	to compute $C = Q^T * B$ (for real matrices);
<code>unmqr</code>	to compute $C = Q^H * B$ (for complex matrices);
<code>trsm</code> (a BLAS routine)	to solve $R^*X = C$.

(The columns of the computed X are the permuted least squares solution vectors x ; the output array *jpvt* specifies the permutation order.)

To compute the elements of Q explicitly, call

<code>orgqr</code>	(for real matrices)
<code>ungqr</code>	(for complex matrices).

?geqp3

Computes the QR factorization of a general m -by- n matrix with column pivoting using level 3 BLAS.

Syntax**FORTRAN 77:**

```
call sgeqp3(m, n, a, lda, jpvt, tau, work, lwork, info)
call dgeqp3(m, n, a, lda, jpvt, tau, work, lwork, info)
call cgeqp3(m, n, a, lda, jpvt, tau, work, lwork, rwork, info)
call zgeqp3(m, n, a, lda, jpvt, tau, work, lwork, rwork, info)
```

Fortran 95:

```
call geqp3(a, jpvt [,tau] [,info])
```

C:

```
lapack_int LAPACKE_sgeqp3 (int matrix_layout, lapack_int m, lapack_int n, float * a,
lapack_int lda, lapack_int * jpvt, float * tau);

lapack_int LAPACKE_dgeqp3 (int matrix_layout, lapack_int m, lapack_int n, double * a,
lapack_int lda, lapack_int * jpvt, double * tau);

lapack_int LAPACKE_cgeqp3 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float * a, lapack_int lda, lapack_int * jpvt, lapack_complex_float * tau);

lapack_int LAPACKE_zgeqp3 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double * a, lapack_int lda, lapack_int * jpvt, lapack_complex_double * tau);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the QR factorization of a general m -by- n matrix A with column pivoting: $A \cdot P = Q \cdot R$ (see [Orthogonal Factorizations](#)) using Level 3 BLAS. Here P denotes an n -by- n permutation matrix. Use this routine instead of [geqpf](#) for better performance.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>m</i>	INTEGER. The number of rows in the matrix A ($m \geq 0$).

<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>a, work</i>	<p>REAL for sgeqp3</p> <p>DOUBLE PRECISION for dgeqp3</p> <p>COMPLEX for cgeqp3</p> <p>DOUBLE COMPLEX for zgeqp3.</p>
	Arrays:
	<i>a</i> (<i>lda,*</i>) contains the matrix <i>A</i> .
	The second dimension of <i>a</i> must be at least $\max(1, n)$.
	<i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array; must be at least $\max(1, 3*n+1)$ for real flavors, and at least $\max(1, n+1)$ for complex flavors.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See <i>Application Notes</i> below for details.</p>
<i>jpvt</i>	<p>INTEGER.</p> <p>Array, size at least $\max(1, n)$.</p> <p>On entry, if $jpvt(i) \neq 0$, the <i>i</i>-th column of <i>A</i> is moved to the beginning of <i>AP</i> before the computation, and fixed in place during the computation.</p> <p>If $jpvt(i) = 0$, the <i>i</i>-th column of <i>A</i> is a free column (that is, it may be interchanged during the computation with any other free column).</p>
<i>rwork</i>	<p>REAL for cgeqp3</p> <p>DOUBLE PRECISION for zgeqp3.</p> <p>A workspace array, size at least $\max(1, 2*n)$. Used in complex flavors only.</p>

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: The elements on and above the diagonal of the array contain the $\min(m, n)$ -by- <i>n</i> upper trapezoidal matrix <i>R</i> (<i>R</i> is upper triangular if $m \geq n$); the elements below the diagonal, with the array <i>tau</i> , present the orthogonal matrix <i>Q</i> as a product of $\min(m, n)$ elementary reflectors (see Orthogonal Factorizations).
<i>tau</i>	<p>REAL for sgeqp3</p> <p>DOUBLE PRECISION for dgeqp3</p> <p>COMPLEX for cgeqp3</p> <p>DOUBLE COMPLEX for zgeqp3.</p> <p>Array, size at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors for the matrix <i>Q</i>.</p>

<i>jpvt</i>	Overwritten by details of the permutation matrix P in the factorization $A^*P = Q^*R$. More precisely, the columns of AP are the columns of A in the following order: $jpvt(1), jpvt(2), \dots, jpvt(n)$.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geqp3` interface are the following:

<i>a</i>	Holds the matrix A of size (m,n) .
<i>jpvt</i>	Holds the vector of length n .
<i>tau</i>	Holds the vector of length $\min(m,n)$

Application Notes

To solve a set of least squares problems minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

?geqp3 (this routine)	to factorize $A^*P = Q^*R$;
<code>ormqr</code>	to compute $C = Q^T B$ (for real matrices);
<code>unmqr</code>	to compute $C = Q^H B$ (for complex matrices);
<code>trsm</code> (a BLAS routine)	to solve $R^*X = C$.

(The columns of the computed X are the permuted least squares solution vectors x ; the output array *jpvt* specifies the permutation order.)

To compute the elements of Q explicitly, call

<code>orgqr</code>	(for real matrices)
<code>ungqr</code>	(for complex matrices).

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?orgqr

Generates the real orthogonal matrix Q of the QR factorization formed by ?geqrf.

Syntax**FORTRAN 77:**

```
call sorgqr(m, n, k, a, lda, tau, work, lwork, info)
call dorgqr(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgqr(a, tau [,info])
```

C:

```
lapack_int LAPACKE_<?>orgqr( int matrix_layout, lapack_int m, lapack_int n, lapack_int k, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine generates the whole or part of m -by- m orthogonal matrix Q of the QR factorization formed by the routines **geqrf/geqrf** or **geqpf/geqpf**. Use this routine after a call to **sgeqrf/dgeqrf** or **sgeqpf/dgeqpf**.

Usually Q is determined from the QR factorization of an m by p matrix A with $m \geq p$. To compute the whole matrix Q , use:

```
call ?orgqr(m, m, p, a, lda, tau, work, lwork, info)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
call ?orgqr(m, p, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the QR factorization of leading k columns of the matrix A :

```
call ?orgqr(m, m, k, a, lda, tau, work, lwork, info)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by leading k columns of the matrix A):

```
call ?orgqr(m, k, k, a, lda, tau, work, lwork, info)
```

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the **C Interface Conventions** section for the C interface principal conventions and type definitions.

m INTEGER. The order of the orthogonal matrix Q ($m \geq 0$).

n INTEGER. The number of columns of Q to be computed
($0 \leq n \leq m$).

k INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a, tau, work REAL for sorgqr

DOUBLE PRECISION for dorgqr

Arrays:

a(*lda,**) and *tau*(*) are the arrays returned by sgqr / dgqr or sgqpf / dgqpf.

The second dimension of *a* must be at least $\max(1, n)$.

The dimension of *tau* must be at least $\max(1, k)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

lwork INTEGER. The size of the *work* array ($lwork \geq n$).

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten by *n* leading columns of the m -by- m orthogonal matrix Q .

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *orgqr* interface are the following:

a Holds the matrix *A* of size (*m,n*).

tau Holds the vector of length (*k*)

Application Notes

For better performance, try using *lwork* = *n***blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed *Q* differs from an exactly orthogonal matrix by a matrix *E* such that

$$\|E\|_2 = O(\epsilon) * \|A\|_2 \text{ where } \epsilon \text{ is the machine precision.}$$

The total number of floating-point operations is approximately $4*m*n*k - 2*(m + n)*k^2 + (4/3)*k^3$.

If $n = k$, the number is approximately $(2/3)*n^2*(3m - n)$.

The complex counterpart of this routine is [ungqr](#).

?ormqr

Multiples a real matrix by the orthogonal matrix Q of the QR factorization formed by ?geqrf or ?geqpf.

Syntax

FORTRAN 77:

```
call sormqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormqr(a, tau, c [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>ormqr( int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const <datatype>* a, lapack_int lda, const <datatype>*
tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine multiplies a real matrix *C* by *Q* or *Q*^T, where *Q* is the orthogonal matrix *Q* of the *QR* factorization formed by the routines [geqrf/geqrf](#) or [geqpf/geqpf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products *Q***C*, *Q*^T**C*, *C***Q*, or *C***Q*^T (overwriting the result on *C*).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

side	CHARACTER*1. Must be either 'L' or 'R'.
------	---

If $\text{side} = \text{'L'}$, Q or Q^T is applied to C from the left.

If $\text{side} = \text{'R'}$, Q or Q^T is applied to C from the right.

trans CHARACTER*1. Must be either 'N' or 'T'.

If $\text{trans} = \text{'N'}$, the routine multiplies C by Q .

If $\text{trans} = \text{'T'}$, the routine multiplies C by Q^T .

m INTEGER. The number of rows in the matrix C ($m \geq 0$).

n INTEGER. The number of columns in C ($n \geq 0$).

k INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints:

$0 \leq k \leq m$ if $\text{side} = \text{'L'}$;

$0 \leq k \leq n$ if $\text{side} = \text{'R'}$.

a, τ, c, work

REAL for sgeqr

DOUBLE PRECISION for dgeqr.

Arrays:

$a(\text{lda},*)$ and $\tau(*)$ are the arrays returned by sgeqr / dgeqr or sgeqpf / dgeqpf. The second dimension of a must be at least $\max(1, k)$. The dimension of τ must be at least $\max(1, k)$.

$c(\text{ldc},*)$ contains the matrix C .

The second dimension of c must be at least $\max(1, n)$

work is a workspace array, its dimension $\max(1, \text{lwork})$.

lda INTEGER. The leading dimension of a . Constraints:

$\text{lda} \geq \max(1, m)$ if $\text{side} = \text{'L'}$;

$\text{lda} \geq \max(1, n)$ if $\text{side} = \text{'R'}$.

ldc INTEGER. The leading dimension of c . Constraint:

$\text{ldc} \geq \max(1, m)$.

lwork INTEGER. The size of the work array. Constraints:

$\text{lwork} \geq \max(1, n)$ if $\text{side} = \text{'L'}$;

$\text{lwork} \geq \max(1, m)$ if $\text{side} = \text{'R'}$.

If $\text{lwork} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the work array, returns this value as the first entry of the work array, and no error message related to lwork is issued by `xerbla`.

See *Application Notes* for the suggested value of lwork .

Output Parameters

c Overwritten by the product Q^*C , $Q^{T*}C$, C^*Q , or C^*Q^T (as specified by side and trans).

<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormqr` interface are the following:

<code>a</code>	Holds the matrix A of size (r,k) . <code>r = m</code> if <code>side = 'L'</code> . <code>r = n</code> if <code>side = 'R'</code> .
<code>tau</code>	Holds the vector of length (k) .
<code>c</code>	Holds the matrix C of size (m,n) .
<code>side</code>	Must be ' <code>L</code> ' or ' <code>R</code> '. The default value is ' <code>L</code> '.
<code>trans</code>	Must be ' <code>N</code> ' or ' <code>T</code> '. The default value is ' <code>N</code> '.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [unmqr](#).

?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by [?geqrf](#).

Syntax

FORTRAN 77:

```
call cungqr(m, n, k, a, lda, tau, work, lwork, info)
```

```
call zungqr(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungqr(a, tau [,info])
```

C:

```
lapack_int LAPACKE_<?>ungqr( int matrix_layout, lapack_int m, lapack_int n, lapack_int k, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine generates the whole or part of m -by- m unitary matrix Q of the QR factorization formed by the routines [geqrf/geqrf](#) or [geqpf/geqpf](#). Use this routine after a call to [cgeqrf/zgeqrf](#) or [cgeqpf/zgeqpf](#).

Usually Q is determined from the QR factorization of an m by p matrix A with $m \geq p$. To compute the whole matrix Q , use:

```
call ?ungqr(m, m, p, a, lda, tau, work, lwork, info)
```

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the columns of A):

```
call ?ungqr(m, p, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the QR factorization of the leading k columns of the matrix A :

```
call ?ungqr(m, m, k, a, lda, tau, work, lwork, info)
```

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by the leading k columns of the matrix A):

```
call ?ungqr(m, k, k, a, lda, tau, work, lwork, info)
```

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m INTEGER. The order of the unitary matrix Q ($m \geq 0$).

n INTEGER. The number of columns of Q to be computed ($0 \leq n \leq m$).

k INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq n$).

a, tau, work COMPLEX for cungqr
DOUBLE COMPLEX for zungqr
Arrays: *a*(*lda,**) and *tau*(*) are the arrays returned by [cgeqrf/zgeqrf](#) or [cgeqpf/zgeqpf](#).
The second dimension of *a* must be at least $\max(1, n)$.

The dimension of *tau* must be at least $\max(1, k)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

lwork INTEGER. The size of the *work* array ($lwork \geq n$).

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten by *n* leading columns of the *m*-by-*m* unitary matrix *Q*.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ungqr` interface are the following:

a Holds the matrix *A* of size (*m,n*).

tau Holds the vector of length (*k*).

Application Notes

For better performance, try using *lwork* = *n*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed *Q* differs from an exactly unitary matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $16*m*n*k - 8*(m + n)*k^2 + (16/3)*k^3$.

If $n = k$, the number is approximately $(8/3) * n^2 * (3m - n)$.

The real counterpart of this routine is [orgqr](#).

?unmqr

Multiples a complex matrix by the unitary matrix Q of the QR factorization formed by ?geqrf.

Syntax

FORTRAN 77:

```
call cunmqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmqr(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmqr(a, tau, c [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>unmqr( int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const <datatype>* a, lapack_int lda, const <datatype>*
tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine multiplies a rectangular complex matrix C by Q or Q^H , where Q is the unitary matrix Q of the QR factorization formed by the routines [geqrf/geqrf](#) or [geqpf/geqpf](#).

Depending on the parameters $side$ and $trans$, the routine can form one of the matrix products Q^*C , $Q^{H*}C$, C^*Q , or C^*Q^H (overwriting the result on C).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).

<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a</i> , <i>c</i> , <i>tau</i> , <i>work</i>	COMPLEX for <i>cgeqrf</i> DOUBLE COMPLEX for <i>zgeqrf</i> . Arrays: <i>a</i> (<i>lda</i> ,*) and <i>tau</i> (*) are the arrays returned by <i>cgeqrf</i> / <i>zgeqrf</i> or <i>cgeqpf</i> / <i>zgeqpf</i> . The second dimension of <i>a</i> must be at least $\max(1, k)$. The dimension of <i>tau</i> must be at least $\max(1, k)$. <i>c</i> (<i>ldc</i> ,*) contains the matrix <i>C</i> . The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> . Constraints: <i>lda</i> $\geq \max(1, m)$ if <i>side</i> = 'L'; <i>lda</i> $\geq \max(1, n)$ if <i>side</i> = 'R'.
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> . Constraint: <i>ldc</i> $\geq \max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints: <i>lwork</i> $\geq \max(1, n)$ if <i>side</i> = 'L'; <i>lwork</i> $\geq \max(1, m)$ if <i>side</i> = 'R'. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , Q^H*C , C^*Q , or C^*Q^H (as specified by <i>side</i> and <i>trans</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmqr` interface are the following:

<code>a</code>	Holds the matrix A of size (r,k) . $r = m$ if <code>side = 'L'</code> . $r = n$ if <code>side = 'R'</code> .
<code>tau</code>	Holds the vector of length (k) .
<code>c</code>	Holds the matrix C of size (m,n) .
<code>side</code>	Must be ' <code>L</code> ' or ' <code>R</code> '. The default value is ' <code>L</code> '.
<code>trans</code>	Must be ' <code>N</code> ' or ' <code>C</code> '. The default value is ' <code>N</code> '.

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [ormqr](#).

?gelqf

Computes the LQ factorization of a general m -by- n matrix.

Syntax

FORTRAN 77:

```
call sgelqf(m, n, a, lda, tau, work, lwork, info)
call dgelqf(m, n, a, lda, tau, work, lwork, info)
call cgelqf(m, n, a, lda, tau, work, lwork, info)
call zgelqf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call gelqf(a [, tau] [,info])
```

C:

```
lapack_int LAPACKE_<?>gelqf( int matrix_layout, lapack_int m, lapack_int n, <datatype>* a, lapack_int lda, <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the LQ factorization of a general m -by- n matrix A (see [Orthogonal Factorizations](#)). No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ *elementary reflectors*. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>m</code>	INTEGER. The number of rows in the matrix A ($m \geq 0$).
<code>n</code>	INTEGER. The number of columns in A ($n \geq 0$).
<code>a, work</code>	REAL for sgelqf DOUBLE PRECISION for dgelqf COMPLEX for cgelqf DOUBLE COMPLEX for zgelqf. Arrays: <code>a(lda,*)</code> contains the matrix A . The second dimension of <code>a</code> must be at least $\max(1, n)$. <code>work</code> is a workspace array, its dimension $\max(1, lwork)$.
<code>lda</code>	INTEGER. The leading dimension of <code>a</code> ; at least $\max(1, m)$.
<code>lwork</code>	INTEGER. The size of the <code>work</code> array; at least $\max(1, m)$. If <code>lwork = -1</code> , then a workspace query is assumed; the routine only calculates the optimal size of the <code>work</code> array, returns this value as the first entry of the <code>work</code> array, and no error message related to <code>lwork</code> is issued by <code>xerbla</code> . See <i>Application Notes</i> for the suggested value of <code>lwork</code> .

Output Parameters

<code>a</code>	Overwritten by the factorization data as follows:
----------------	---

If $m \leq n$, the elements above the diagonal are overwritten by the details of the unitary (orthogonal) matrix Q , and the lower triangle is overwritten by the corresponding elements of the lower triangular matrix L .

If $m > n$, the strictly upper triangular part is overwritten by the details of the matrix Q , and the remaining elements are overwritten by the corresponding elements of the m -by- n lower trapezoidal matrix L .

tau

REAL for sgelqf

DOUBLE PRECISION for dgelqf

COMPLEX for cgelqf

DOUBLE COMPLEX for zgelqf.

Array, DIMENSION at least $\max(1, \min(m, n))$.Contains additional information on the matrix Q .*work(1)*If $info = 0$, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.*info*

INTEGER.

If $info = 0$, the execution is successful.If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gelqf* interface are the following:

a Holds the matrix A of size (m,n) .

tau Holds the vector of length $\min(m,n)$.

Application Notes

For better performance, try using *lwork* = $m * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed factorization is the exact factorization of a matrix $A + E$, where

$$\|E\|_2 = O(\varepsilon) \|A\|_2.$$

The approximate number of floating-point operations for real flavors is

$$(4/3)n^3 \quad \text{if } m = n,$$

$$(2/3)n^2(3m-n) \quad \text{if } m > n,$$

$$(2/3)m^2(3n-m) \quad \text{if } m < n.$$

The number of operations for complex flavors is 4 times greater.

To find the minimum-norm solution of an underdetermined least squares problem minimizing $\|A^*x - b\|_2$ for all columns b of a given matrix B , you can call the following:

?gelqf (this routine)	to factorize $A = L^*Q$;
trsm (a BLAS routine)	to solve $L^*Y = B$ for Y ;
ormlq	to compute $X = (Q_1)^T Y$ (for real matrices);
unmlq	to compute $X = (Q_1)^H Y$ (for complex matrices).

(The columns of the computed X are the minimum-norm solution vectors x . Here A is an m -by- n matrix with $m < n$; Q_1 denotes the first m columns of Q).

To compute the elements of Q explicitly, call

orglq	(for real matrices)
unglq	(for complex matrices).

See Also

[mkl_progress](#)

?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by ?gelqf.

Syntax

FORTRAN 77:

```
call sorglq(m, n, k, a, lda, tau, work, lwork, info)
call dorglq(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orglq(a, tau [,info])
```

C:

```
lapack_int LAPACKE_<?>orglq( int matrix_layout, lapack_int m, lapack_int n, lapack_int k, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine generates the whole or part of n -by- n orthogonal matrix Q of the LQ factorization formed by the routines [gelqf/gelqf](#). Use this routine after a call to [sgelqf/dgelqf](#).

Usually Q is determined from the LQ factorization of an p -by- n matrix A with $n \geq p$. To compute the whole matrix Q , use:

```
call ?orglq(n, n, p, a, lda, tau, work, lwork, info)
```

To compute the leading p rows of Q , which form an orthonormal basis in the space spanned by the rows of A , use:

```
call ?orglq(p, n, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the LQ factorization of the leading k rows of A , use:

```
call ?orglq(n, n, k, a, lda, tau, work, lwork, info)
```

To compute the leading k rows of Q^k , which form an orthonormal basis in the space spanned by the leading k rows of A , use:

```
call ?orgqr(k, n, k, a, lda, tau, work, lwork, info)
```

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m INTEGER. The number of rows of Q to be computed
 $(0 \leq m \leq n)$.

n INTEGER. The order of the orthogonal matrix Q ($n \geq m$).

k INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq m$).

a, tau, work REAL for `sorglq`
 DOUBLE PRECISION for `dorglq`
 Arrays: *a*(*lda*,*) and *tau*(*) are the arrays returned by [sgelqf/dgelqf](#).

The second dimension of *a* must be at least $\max(1, n)$.

The dimension of *tau* must be at least $\max(1, k)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

lwork INTEGER. The size of the *work* array; at least $\max(1, m)$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a Overwritten by *m* leading rows of the n -by- n orthogonal matrix Q .

<code>work(1)</code>	If <code>info = 0</code> , on exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance. Use this <code>lwork</code> for subsequent runs.
<code>info</code>	INTEGER. If <code>info = 0</code> , the execution is successful. If <code>info = -i</code> , the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `orglq` interface are the following:

<code>a</code>	Holds the matrix A of size (m,n) .
<code>tau</code>	Holds the vector of length (k) .

Application Notes

For better performance, try using `lwork = m*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $4*m*n*k - 2*(m + n)*k^2 + (4/3)*k^3$.

If $m = k$, the number is approximately $(2/3)*m^2*(3n - m)$.

The complex counterpart of this routine is [unglq](#).

?ormlq

Multiples a real matrix by the orthogonal matrix Q of the LQ factorization formed by ?gelqf.

Syntax

FORTRAN 77:

```
call sormlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormlq(a, tau, c [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_sormlq( int matrix_layout, char side, char trans, lapack_int m,
                           lapack_int n, lapack_int k, const <datatype>* a, lapack_int lda, const <datatype>* tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the orthogonal matrix Q of the LQ factorization formed by the routine [gelqf/gelqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , Q^T*C , C^*Q , or C^*Q^T (overwriting the result on C).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a</i> , <i>c</i> , <i>tau</i> , <i>work</i>	REAL for sormlq DOUBLE PRECISION for dormlq. Arrays: <i>a</i> (<i>lda</i> ,*) and <i>tau</i> (*) are arrays returned by ?gelqf. The second dimension of <i>a</i> must be: at least $\max(1, m)$ if <i>side</i> = 'L'; at least $\max(1, n)$ if <i>side</i> = 'R'. The dimension of <i>tau</i> must be at least $\max(1, k)$.

c(*ldc*,*) contains the matrix *C*.

The second dimension of *c* must be at least $\max(1, n)$

work is a workspace array, its dimension $\max(1, \text{lwork})$.

lda

INTEGER. The leading dimension of *a*; *lda* $\geq \max(1, k)$.

ldc

INTEGER. The leading dimension of *c*; *ldc* $\geq \max(1, m)$.

lwork

INTEGER. The size of the *work* array. Constraints:

lwork $\geq \max(1, n)$ if *side* = 'L';

lwork $\geq \max(1, m)$ if *side* = 'R'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c

Overwritten by the product Q^*C , $Q^{T*}C$, C^*Q , or C^*Q^T (as specified by *side* and *trans*).

work(1)

If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *ormlq* interface are the following:

a Holds the matrix *A* of size (*k,m*).

tau Holds the vector of length (*k*).

c Holds the matrix *C* of size (*m,n*).

side Must be 'L' or 'R'. The default value is 'L'.

trans Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n***blocksize* (if *side* = 'L') or *lwork* = *m***blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork= -1*.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork= -1*, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [unmlq](#).

?unglq

Generates the complex unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

FORTRAN 77:

```
call cunglq(m, n, k, a, lda, tau, work, lwork, info)
call zunglq(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call unglq(a, tau [,info])
```

C:

```
lapack_int LAPACKE_<?>unglq( int matrix_layout, lapack_int m, lapack_int n, lapack_int k, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine generates the whole or part of n -by- n unitary matrix Q of the *LQ* factorization formed by the routines [gelqf/gelqf](#). Use this routine after a call to [cgelqf/zgelqf](#).

Usually Q is determined from the *LQ* factorization of an p -by- n matrix A with $n < p$. To compute the whole matrix Q , use:

```
call ?unglq(n, n, p, a, lda, tau, work, lwork, info)
```

To compute the leading p rows of Q , which form an orthonormal basis in the space spanned by the rows of A , use:

```
call ?unglq(p, n, p, a, lda, tau, work, lwork, info)
```

To compute the matrix Q^k of the *LQ* factorization of the leading k rows of the matrix A , use:

```
call ?unglq(n, n, k, a, lda, tau, work, lwork, info)
```

To compute the leading k rows of Q^k , which form an orthonormal basis in the space spanned by the leading k rows of the matrix A , use:

```
call ?ungqr(k, n, k, a, lda, tau, work, lwork, info)
```

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows of Q to be computed ($0 \leq m \leq n$).
<i>n</i>	INTEGER. The order of the unitary matrix Q ($n \geq m$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($0 \leq k \leq m$).
<i>a, tau, work</i>	<p>COMPLEX for <code>cunglq</code> DOUBLE COMPLEX for <code>zunglq</code></p> <p>Arrays: <i>a</i>(<i>lda</i>,*) and <i>tau</i>(*) are the arrays returned by <code>sgelqf/dgelqf</code>.</p> <p>The second dimension of <i>a</i> must be at least <code>max(1, n)</code>.</p> <p>The dimension of <i>tau</i> must be at least <code>max(1, k)</code>.</p> <p><i>work</i> is a workspace array, its dimension <code>max(1, lwork)</code>.</p>
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least <code>max(1, m)</code> .
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array; at least <code>max(1, m)</code>.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code>.</p> <p>See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>

Output Parameters

<i>a</i>	Overwritten by <i>m</i> leading rows of the n -by- n unitary matrix Q .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p>

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unglq` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m,n</i>).
<i>tau</i>	Holds the vector of length (<i>k</i>).

Application Notes

For better performance, try using `lwork = m*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to `-1`, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Q differs from an exactly unitary matrix by a matrix E such that $\|E\|_2 = O(\varepsilon) * \|A\|_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $16*m*n*k - 8*(m + n)*k^2 + (16/3)*k^3$.

If $m = k$, the number is approximately $(8/3)*m^2*(3n - m)$.

The real counterpart of this routine is [orglq](#).

?unmlq

Multiplies a complex matrix by the unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

FORTRAN 77:

```
call cunmlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmlq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmlq(a, tau, c [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>unmlq( int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const <datatype>* a, lapack_int lda, const <datatype>*
tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine multiplies a real m -by- n matrix C by Q or Q^H , where Q is the unitary matrix Q of the LQ factorization formed by the routine [gelqf/gelqf](#).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (overwriting the result on C).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q_H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a</i> , <i>c</i> , <i>tau</i> , <i>work</i>	COMPLEX for <code>cunmlq</code> DOUBLE COMPLEX for <code>zunmlq</code> . Arrays: <i>a</i> (<i>lda</i> ,*) and <i>tau</i> (*) are arrays returned by <code>?gelqf</code> . The second dimension of <i>a</i> must be: at least $\max(1, m)$ if <i>side</i> = 'L'; at least $\max(1, n)$ if <i>side</i> = 'R'. The dimension of <i>tau</i> must be at least $\max(1, k)$. <i>c</i> (<i>ldc</i> ,*) contains the matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work</i> is a workspace array, its dimension $\max(1, lwork)$. <i>lda</i> INTEGER. The leading dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, k)$. <i>ldc</i> INTEGER. The leading dimension of <i>c</i> ; <i>ldc</i> $\geq \max(1, m)$. <i>lwork</i> INTEGER. The size of the <i>work</i> array. Constraints: <i>lwork</i> $\geq \max(1, n)$ if <i>side</i> = 'L'; <i>lwork</i> $\geq \max(1, m)$ if <i>side</i> = 'R'. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code> .

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>c</i>	Overwritten by the product $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (as specified by <i>side</i> and <i>trans</i>).
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmqlq` interface are the following:

<i>a</i>	Holds the matrix A of size (k,m).
<i>tau</i>	Holds the vector of length (k).
<i>c</i>	Holds the matrix C of size (m,n).
<i>side</i>	Must be ' <code>L</code> ' or ' <code>R</code> '. The default value is ' <code>L</code> '.
<i>trans</i>	Must be ' <code>N</code> ' or ' <code>C</code> '. The default value is ' <code>N</code> '.

Application Notes

For better performance, try using *lwork* = $n*blocksize$ (if *side* = '`L`') or *lwork* = $m*blocksize$ (if *side* = '`R`') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [ormlq](#).

?geqlf

Computes the *QL factorization* of a general m -by- n matrix.

Syntax

FORTRAN 77:

```
call sgeqlf(m, n, a, lda, tau, work, lwork, info)
call dgeqlf(m, n, a, lda, tau, work, lwork, info)
call cgeqlf(m, n, a, lda, tau, work, lwork, info)
call zgeqlf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call geqlf(a [, tau] [,info])
```

C:

```
lapack_int LAPACKE_<?>geqlf( int matrix_layout, lapack_int m, lapack_int n, <datatype>* a, lapack_int lda, <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the QL factorization of a general m -by- n matrix A . No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a, work REAL for sgeqlf

DOUBLE PRECISION for dgeqlf

COMPLEX for cgeqlf

DOUBLE COMPLEX for zgeqlf.

Arrays: *a*(*lda,**) contains the matrix A .

The second dimension of *a* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

*lwork*INTEGER. The size of the *work* array; at least $\max(1, n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a

Overwritten on exit by the factorization data as follows:

If $m \geq n$, the lower triangle of the subarray $a(m-n+1:m, 1:n)$ contains the n -by- n lower triangular matrix L ; if $m \leq n$, the elements on and below the $(n-m)$ -th superdiagonal contain the m -by- n lower trapezoidal matrix L ; in both cases, the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

tau

REAL for sgeqlf

DOUBLE PRECISION for dgeqlf

COMPLEX for cgeqlf

DOUBLE COMPLEX for zgeqlf.

Array, DIMENSION at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors for the matrix Q .

work(1)

If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *geqlf* interface are the following:

*a*Holds the matrix A of size (m,n) .*tau*Holds the vector of length $\min(m,n)$.

Application Notes

For better performance, try using $lwork = n * \text{blocksize}$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1 .

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

orgql	to generate matrix Q (for real matrices);
ungql	to generate matrix Q (for complex matrices);
ormql	to apply matrix Q (for real matrices);
unmql	to apply matrix Q (for complex matrices).

See Also

[mkl_progress](#)

?orgql

Generates the real matrix *Q* of the *QL* factorization formed by ?geqlf.

Syntax

FORTRAN 77:

```
call sorgql(m, n, k, a, lda, tau, work, lwork, info)
call dorgql(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgql(a, tau [,info])
```

C:

```
lapack_int LAPACKE_<?>orgql( int matrix_layout, lapack_int m, lapack_int n, lapack_int k, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: [mkl.fi](#)
- Fortran 95: [lapack.f90](#)
- C: [mkl.h](#)

Description

The routine generates an *m*-by-*n* real matrix *Q* with orthonormal columns, which is defined as the last *n* columns of a product of *k* elementary reflectors *H*(*i*) of order *m*: $Q = H(k) * \dots * H(2) * H(1)$ as returned by the routines [geqlf](#)/[dgeqlf](#). Use this routine after a call to [sgeqlf](#)/[dgeqlf](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows of the matrix Q ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix Q ($m \geq n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<i>a</i> , <i>tau</i> , <i>work</i>	<p>REAL for <code>sorgql</code> DOUBLE PRECISION for <code>dorgql</code></p> <p>Arrays: $a(lda, *), tau(*)$.</p> <p>On entry, the $(n - k + i)$th column of <i>a</i> must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>sgeqlf/dgeqlf</code> in the last <i>k</i> columns of its array argument <i>a</i>; <i>tau(i)</i> must contain the scalar factor of the elementary reflector $H(i)$, as returned by <code>sgeqlf/dgeqlf</code>;</p> <p>The second dimension of <i>a</i> must be at least $\max(1, n)$.</p> <p>The dimension of <i>tau</i> must be at least $\max(1, k)$.</p> <p><i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array; at least $\max(1, n)$.</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code>.</p> <p>See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>

Output Parameters

<i>a</i>	Overwritten by the m -by- n matrix Q .
<i>work(1)</i>	If <i>info</i> = 0 , on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = $-i$, the <i>i</i>-th parameter had an illegal value.</p>

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `orgql` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (m, n) .
<i>tau</i>	Holds the vector of length (k) .

Application Notes

For better performance, try using `lwork =n*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [ungql](#).

?ungql

Generates the complex matrix Q of the QL factorization formed by ?geqlf.

Syntax

FORTRAN 77:

```
call cungql(m, n, k, a, lda, tau, work, lwork, info)
call zungql(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungql(a, tau [,info])
```

C:

```
lapack_int LAPACKE_<?>ungql( int matrix_layout, lapack_int m, lapack_int n, lapack_int
k, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine generates an m -by- n complex matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors $H(i)$ of order m : $Q = H(k) * \dots * H(2) * H(1)$ as returned by the routines [geqlf](#)/[zgeqlf](#). Use this routine after a call to [cgeqlf](#)/[zgeqlf](#).

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

`m`

INTEGER. The number of rows of the matrix Q ($m \geq 0$).

<i>n</i>	INTEGER. The number of columns of the matrix Q ($m \geq n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<i>a, tau, work</i>	<p>COMPLEX for <code>cungql</code> DOUBLE COMPLEX for <code>zungql</code></p> <p>Arrays: $a(lda,*), tau(*), work(lwork)$.</p> <p>On entry, the $(n - k + i)$th column of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>cgeqlf/zgeqlf</code> in the last k columns of its array argument a;</p> <p>$tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by <code>cgeqlf/zgeqlf</code>;</p> <p>The second dimension of a must be at least $\max(1, n)$.</p> <p>The dimension of tau must be at least $\max(1, k)$.</p> <p>$work$ is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
<i>lwork</i>	<p>INTEGER. The size of the $work$ array; at least $\max(1, n)$.</p> <p>If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by <code>xerbla</code>.</p> <p>See <i>Application Notes</i> for the suggested value of $lwork$.</p>

Output Parameters

<i>a</i>	Overwritten by the m -by- n matrix Q .
<i>work(1)</i>	If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
<i>info</i>	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p>

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ungql` interface are the following:

<i>a</i>	Holds the matrix A of size (m,n) .
<i>tau</i>	Holds the vector of length (k) .

Application Notes

For better performance, try using `lwork =n*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to `-1`, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is `orgql`.

?ormql

Multiples a real matrix by the orthogonal matrix Q of the QL factorization formed by ?geqlf.

Syntax

FORTRAN 77:

```
call sormql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormql(a, tau, c [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>ormql( int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const <datatype>* a, lapack_int lda, const <datatype>*
tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the orthogonal matrix Q of the *QL* factorization formed by the routine `geqlf/geqlf`.

Depending on the parameters `side` and `trans`, the routine `ormql` can form one of the matrix products Q^*C , $Q^{T*}C$, C^*Q , or C^*Q^T (overwriting the result over C).

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>side</code>	CHARACTER*1. Must be either 'L' or 'R'.
-------------------	---

If $\text{side} = \text{'L'}$, Q or Q^T is applied to C from the left.

If $\text{side} = \text{'R'}$, Q or Q^T is applied to C from the right.

trans

CHARACTER*1. Must be either 'N' or 'T'.

If $\text{trans} = \text{'N'}$, the routine multiplies C by Q .

If $\text{trans} = \text{'T'}$, the routine multiplies C by Q^T .

m

INTEGER. The number of rows in the matrix C ($m \geq 0$).

n

INTEGER. The number of columns in C ($n \geq 0$).

k

INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints:

$0 \leq k \leq m$ if $\text{side} = \text{'L'}$;

$0 \leq k \leq n$ if $\text{side} = \text{'R'}$.

a, τ, c, work

REAL for `sormql`

DOUBLE PRECISION for `dormql`.

Arrays: $a(\text{lda},*)$, $\tau(*)$, $c(\text{ldc},*)$.

On entry, the i th column of a must contain the vector which defines the elementary reflector H_i , for $i = 1, 2, \dots, k$, as returned by `sgeqlf/dgeqlf` in the last k columns of its array argument a .

The second dimension of a must be at least $\max(1, k)$.

$\tau(i)$ must contain the scalar factor of the elementary reflector H_i , as returned by `sgeqlf/dgeqlf`.

The dimension of τ must be at least $\max(1, k)$.

$c(\text{ldc},*)$ contains the m -by- n matrix C .

The second dimension of c must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, \text{lwork})$.

lda

INTEGER. The leading dimension of a ;

if $\text{side} = \text{'L'}$, $\text{lda} \geq \max(1, m)$;

if $\text{side} = \text{'R'}$, $\text{lda} \geq \max(1, n)$.

ldc

INTEGER. The leading dimension of c ; $\text{ldc} \geq \max(1, m)$.

lwork

INTEGER. The size of the work array. Constraints:

$\text{lwork} \geq \max(1, n)$ if $\text{side} = \text{'L'}$;

$\text{lwork} \geq \max(1, m)$ if $\text{side} = \text{'R'}$.

If $\text{lwork} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the work array, returns this value as the first entry of the work array, and no error message related to lwork is issued by `xerbla`.

See *Application Notes* for the suggested value of lwork .

Output Parameters

<i>c</i>	Overwritten by the product $Q^* C$, $Q^T * C$, $C^* Q$, or $C^* Q^T$ (as specified by <i>side</i> and <i>trans</i>).
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormql` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>r,k</i>). <i>r</i> = <i>m</i> if <i>side</i> = 'L'. <i>r</i> = <i>n</i> if <i>side</i> = 'R'.
<i>tau</i>	Holds the vector of length (<i>k</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m,n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (if *side* = 'L') or *lwork* = *m*blocksize* (if *side* = 'R') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is `unmql`.

?unmql

Multiplies a complex matrix by the unitary matrix Q of the QL factorization formed by ?geqlf.

Syntax

FORTRAN 77:

```
call cunmql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmql(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmql(a, tau, c [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>unmql( int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const <datatype>* a, lapack_int lda, const <datatype>*
tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine multiplies a complex m -by- n matrix C by Q or Q^H , where Q is the unitary matrix Q of the QL factorization formed by the routine [geqlf/geqlf](#).

Depending on the parameters *side* and *trans*, the routine [unmql](#) can form one of the matrix products Q^*C , Q^H*C , $C*Q$, or $C*Q^H$ (overwriting the result over C).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$ if <i>side</i> = 'L'; $0 \leq k \leq n$ if <i>side</i> = 'R'.
<i>a</i> , <i>tau</i> , <i>c</i> , <i>work</i>	COMPLEX for cunmql DOUBLE COMPLEX for zunmql .

Arrays: $a(lda, *)$, $\tau(*)$, $c(ldc, *)$, $work(lwork)$.

On entry, the i -th column of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by `cgeqlf/zgeqlf` in the last k columns of its array argument a .

The second dimension of a must be at least $\max(1, k)$.

$\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by `cgeqlf/zgeqlf`.

The dimension of τ must be at least $\max(1, k)$.

$c(ldc, *)$ contains the m -by- n matrix C .

The second dimension of c must be at least $\max(1, n)$

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, n)$.

ldc INTEGER. The leading dimension of c ; $ldc \geq \max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array. Constraints:

$lwork \geq \max(1, n)$ if $side = 'L'$;

$lwork \geq \max(1, m)$ if $side = 'R'$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c Overwritten by the product Q^*C , Q^H*C , C^*Q , or C^*Q^H (as specified by $side$ and $trans$).

$work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmql` interface are the following:

a Holds the matrix A of size (r, k) .

$r = m$ if $side = 'L'$.

$r = n$ if $side = 'R'$.

<i>tau</i>	Holds the vector of length (<i>k</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m,n</i>).
<i>side</i>	Must be ' <i>L</i> ' or ' <i>R</i> '. The default value is ' <i>L</i> '.
<i>trans</i>	Must be ' <i>N</i> ' or ' <i>C</i> '. The default value is ' <i>N</i> '.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (if *side* = '*L*') or *lwork* = *m*blocksize* (if *side* = '*R*') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [ormql](#).

?gerqf

Computes the *RQ* factorization of a general *m*-by-*n* matrix.

Syntax

FORTRAN 77:

```
call sgerqf(m, n, a, lda, tau, work, lwork, info)
call dgerqf(m, n, a, lda, tau, work, lwork, info)
call cgerqf(m, n, a, lda, tau, work, lwork, info)
call zgerqf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call gerqf(a [, tau] [, info])
```

C:

```
lapack_int LAPACKE_<?>gerqf( int matrix_layout, lapack_int m, lapack_int n, <datatype>* a, lapack_int lda, <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the *RQ* factorization of a general *m*-by-*n* matrix *A*. No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with Q in this representation.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in A ($n \geq 0$).
<i>a, work</i>	REAL for sgerqf DOUBLE PRECISION for dgerqf COMPLEX for cgerqf DOUBLE COMPLEX for zgerqf. Arrays: <i>a</i> (<i>lda,*</i>) contains the m -by- n matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$. <i>lda</i> INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$. <i>lwork</i> INTEGER. The size of the <i>work</i> array; $lwork \geq \max(1, m)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See Application Notes for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten on exit by the factorization data as follows: if $m \leq n$, the upper triangle of the subarray $a(1:m, n-m+1:n)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m-n)$ th subdiagonal contain the m -by- n upper trapezoidal matrix R ; in both cases, the remaining elements, with the array <i>tau</i> , represent the orthogonal/unitary matrix Q as a product of $\min(m,n)$ elementary reflectors.
<i>tau</i>	REAL for sgerqf

DOUBLE PRECISION for dgerqf
COMPLEX for cgerqf
DOUBLE COMPLEX for zgerqf.
Array, DIMENSION at least max (1, min(m, n)).
Contains scalar factors of the elementary reflectors for the matrix Q .

work(1)
If $info = 0$, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance.

info
INTEGER.
If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gerqf` interface are the following:

<i>a</i>	Holds the matrix A of size (m, n).
<i>tau</i>	Holds the vector of length $\min(m, n)$.

Application Notes

For better performance, try using $lwork = m * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork = -1*.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork = -1*, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

orgrq	to generate matrix Q (for real matrices);
ungrq	to generate matrix Q (for complex matrices);
ormrq	to apply matrix Q (for real matrices);
unmrq	to apply matrix Q (for complex matrices).

See Also

[mkl_progress](#)

?orgqr

Generates the real matrix Q of the RQ factorization formed by ?gerqf.

Syntax**FORTRAN 77:**

```
call sorgrq(m, n, k, a, lda, tau, work, lwork, info)
call dorgqr(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgrq(a, tau [,info])
```

C:

```
lapack_int LAPACKE_<?>orgrq( int matrix_layout, lapack_int m, lapack_int n, lapack_int k, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine generates an m -by- n real matrix Q with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors $H(i)$ of order n : $Q = H(1) * H(2) * \dots * H(k)$ as returned by the routines gerqf/gerqf. Use this routine after a call to sgerqf/dgerqf.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows of the matrix Q ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix Q ($n \geq m$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
<i>a</i> , <i>tau</i> , <i>work</i>	REAL for sorgrq DOUBLE PRECISION for dorgqr Arrays: <i>a</i> (<i>lda</i> ,*), <i>tau</i> (*). On entry, the $(m - k + i)$ -th row of <i>a</i> must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by sgerqf/dgerqf in the last k rows of its array argument <i>a</i> ; <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$, as returned by sgerqf/dgerqf; The second dimension of <i>a</i> must be at least $\max(1, n)$. The dimension of <i>tau</i> must be at least $\max(1, k)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.

<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; at least $\max(1, m)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> .
	See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the m -by- n matrix Q .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `orgqr` interface are the following:

<i>a</i>	Holds the matrix A of size (m, n).
<i>tau</i>	Holds the vector of length (k).

Application Notes

For better performance, try using *lwork* = $m * \text{blocksize}$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is `ungrq`.

?ungqr

Generates the complex matrix Q of the RQ factorization formed by `?gerqf`.

Syntax

FORTRAN 77:

```
call cungrq(m, n, k, a, lda, tau, work, lwork, info)
call zungrq(m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungrq(a, tau [,info])
```

C:

```
lapack_int LAPACKE_<?>ungrq( int matrix_layout, lapack_int m, lapack_int n, lapack_int k, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine generates an m -by- n complex matrix Q with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors $H(i)$ of order n : $Q = H(1)^H \cdot H(2)^H \cdots \cdot H(k)^H$ as returned by the routines [gerqf/gerqf](#). Use this routine after a call to [sgerqf/dgerqf](#).

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m	INTEGER. The number of rows of the matrix Q ($m \geq 0$).
n	INTEGER. The number of columns of the matrix Q ($n \geq m$).
k	INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
$a, tau, work$	<p>REAL for cungrq DOUBLE PRECISION for zungrq Arrays: $a(lda,*), tau(*), work(lwork)$.</p> <p>On entry, the $(m - k + i)$th row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by sgerqf/dgerqf in the last k rows of its array argument a;</p> <p>$tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by sgerqf/dgerqf;</p> <p>The second dimension of a must be at least $\max(1, n)$.</p> <p>The dimension of tau must be at least $\max(1, k)$.</p> <p>$work$ is a workspace array, its dimension $\max(1, lwork)$.</p>
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
$lwork$	INTEGER. The size of the $work$ array; at least $\max(1, m)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a	Overwritten by the m -by- n matrix Q .
$work(1)$	If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ungrq` interface are the following:

a	Holds the matrix A of size (m,n) .
tau	Holds the vector of length (k) .

Application Notes

For better performance, try using $lwork = m * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1 , then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is `orgrq`.

?ormrq

Multiples a real matrix by the orthogonal matrix Q of the RQ factorization formed by ?gerqf.

Syntax

FORTRAN 77:

```
call sormrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormrq(a, tau, c [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>ormrq( int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const <datatype>* a, lapack_int lda, const <datatype>*
tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the real orthogonal matrix defined as a product of k elementary reflectors $H_i : Q = H_1 H_2 \dots H_k$ as returned by the RQ factorization routine [gerqf/gerqf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , $Q^{T*}C$, C^*Q , or C^*Q^T (overwriting the result over C).

Input Parameters

The data types are given for the Fortran interface. A *<datatype>* placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <i>side</i> = 'L'; $0 \leq k \leq n$, if <i>side</i> = 'R'.
<i>a</i> , <i>tau</i> , <i>c</i> , <i>work</i>	REAL for sormrq DOUBLE PRECISION for dormrq. Arrays: <i>a</i> (<i>lda</i> ,*), <i>tau</i> (*), <i>c</i> (<i>ldc</i> ,*). On entry, the <i>i</i> th row of <i>a</i> must contain the vector which defines the elementary reflector H_i , for $i = 1, 2, \dots, k$, as returned by <i>sgerqf/dgerqf</i> in the last k rows of its array argument <i>a</i> .

The second dimension of a must be at least $\max(1, m)$ if $\text{side} = \text{'L'}$, and at least $\max(1, n)$ if $\text{side} = \text{'R'}$.

$\tau_{\text{au}}(i)$ must contain the scalar factor of the elementary reflector H_i , as returned by `sgerqf/dgerqf`.

The dimension of τ_{au} must be at least $\max(1, k)$.

$c(ldc,*)$ contains the m -by- n matrix C .

The second dimension of c must be at least $\max(1, n)$

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, k)$.

ldc INTEGER. The leading dimension of c ; $ldc \geq \max(1, m)$.

$lwork$ INTEGER. The size of the $work$ array. Constraints:

$lwork \geq \max(1, n)$ if $\text{side} = \text{'L'}$;

$lwork \geq \max(1, m)$ if $\text{side} = \text{'R'}$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c Overwritten by the product Q^*C , $Q^{T*}C$, C^*Q , or C^*Q^T (as specified by side and trans).

$work(1)$ If $\text{info} = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormrq` interface are the following:

a Holds the matrix A of size (k,m) .

τ_{au} Holds the vector of length (k) .

c Holds the matrix C of size (m,n) .

$side$ Must be 'L' or 'R' . The default value is 'L' .

$trans$ Must be 'N' or 'T' . The default value is 'N' .

Application Notes

For better performance, try using `lwork = n*blocksize` (if `side = 'L'`) or `lwork = m*blocksize` (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [unmrq](#).

?unmrq

Multiplies a complex matrix by the unitary matrix Q of the RQ factorization formed by ?gerqf.

Syntax

FORTRAN 77:

```
call cunmrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmrq(side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmrq(a, tau, c [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>unmrq( int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const <datatype>* a, lapack_int lda, const <datatype>*
tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine multiplies a complex m -by- n matrix C by Q or Q^H , where Q is the complex unitary matrix defined as a product of k elementary reflectors $H(i)$ of order n : $Q = H(1)H^* H(2)H^* \dots H(k)H$ has returned by the RQ factorization routine [gerqf/gerqf](#).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products Q^*C , $Q^{H*}C$, C^*Q , or C^*Q^H (overwriting the result over C).

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <i>side</i> = 'L'; $0 \leq k \leq n$, if <i>side</i> = 'R'.
<i>a</i> , <i>tau</i> , <i>c</i> , <i>work</i>	COMPLEX for cunmqr DOUBLE COMPLEX for zunmqr. Arrays: $a(lda,*), tau(*), c ldc,*), work(lwork)$. On entry, the <i>i</i> th row of <i>a</i> must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by cgerqf/zgerqf in the last <i>k</i> rows of its array argument <i>a</i> . The second dimension of <i>a</i> must be at least $\max(1, m)$ if <i>side</i> = 'L', and at least $\max(1, n)$ if <i>side</i> = 'R'. <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$, as returned by cgerqf/zgerqf. The dimension of <i>tau</i> must be at least $\max(1, k)$. <i>c</i> (<i>ldc</i> ,*) contains the <i>m</i> -by- <i>n</i> matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work</i> is a workspace array, its dimension $\max(1, lwork)$. <i>lda</i> INTEGER. The leading dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, k)$. <i>ldc</i> INTEGER. The leading dimension of <i>c</i> ; <i>ldc</i> $\geq \max(1, m)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints: $lwork \geq \max(1, n)$ if <i>side</i> = 'L'; $lwork \geq \max(1, m)$ if <i>side</i> = 'R'. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla. See Application Notes for the suggested value of <i>lwork</i> .

Output Parameters

<i>c</i>	Overwritten by the product $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (as specified by <i>side</i> and <i>trans</i>).
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmraq` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>k,m</i>).
<i>tau</i>	Holds the vector of length (<i>k</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m,n</i>).
<i>side</i>	Must be ' <i>L</i> ' or ' <i>R</i> '. The default value is ' <i>L</i> '.
<i>trans</i>	Must be ' <i>N</i> ' or ' <i>C</i> '. The default value is ' <i>N</i> '.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (if *side* = '*L*') or *lwork* = *m*blocksize* (if *side* = '*R*') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is `ormrq`.

?tzrzf

Reduces the upper trapezoidal matrix *A* to upper triangular form.

Syntax

FORTRAN 77:

```
call stzrzf(m, n, a, lda, tau, work, lwork, info)
```

```
call dtzrzf(m, n, a, lda, tau, work, lwork, info)
call ctzrzf(m, n, a, lda, tau, work, lwork, info)
call ztzrzf(m, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call tzrzf(a [, tau] [,info])
```

C:

```
lapack_int LAPACKE_<?>tzrzf( int matrix_layout, lapack_int m, lapack_int n, <datatype>* a, lapack_int lda, <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces the m -by- n ($m \leq n$) real/complex upper trapezoidal matrix A to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix $A = [A1 A2] = [A(1:m, 1:m), A(1:m, m+1:n)]$ is factored as

$$A = [R \ O] * Z,$$

where Z is an n -by- n orthogonal/unitary matrix, R is an m -by- m upper triangular matrix, and O is the m -by-($n-m$) zero matrix.

See [larz](#) that applies an elementary reflector returned by `?tzrzf` to a general matrix.

The `?tzrzf` routine replaces the deprecated `?tzrqf` routine.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq m$).

a, work REAL for stzrzf

DOUBLE PRECISION for dtzrzf

COMPLEX for ctzrzf

DOUBLE COMPLEX for ztzrzf.

Arrays: $a(lda,*)$, $work(lwork)$. The leading m -by- n upper trapezoidal part of the array a contains the matrix A to be factorized.

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

lwork INTEGER. The size of the $work$ array;

lwork $\geq \max(1, m)$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>a</i>	Overwritten on exit by the factorization data as follows: the leading <i>m</i> -by- <i>m</i> upper triangular part of <i>a</i> contains the upper triangular matrix <i>R</i> , and elements <i>m</i> + 1 to <i>n</i> of the first <i>m</i> rows of <i>a</i> , with the array <i>tau</i> , represent the orthogonal matrix <i>Z</i> as a product of <i>m</i> elementary reflectors.
<i>tau</i>	REAL for <i>stzrzf</i> DOUBLE PRECISION for <i>dtzrzf</i> COMPLEX for <i>ctzrzf</i> DOUBLE COMPLEX for <i>ztzrzf</i> . Array, DIMENSION at least $\max(1, m)$. Contains scalar factors of the elementary reflectors for the matrix <i>Z</i> .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *tzrzf* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m,n</i>).
<i>tau</i>	Holds the vector of length (<i>m</i>).

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, *Z*(*k*), which is used to introduce zeros into the (*m* - *k* + 1)-th row of *A*, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix}$$

where for real flavors

$$T(k) = I - tau u(k)^* u(k)^T, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

and for complex flavors

$$T(k) = I - tau u(k)^* u(k)^H, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

tau is a scalar and *z(k)* is an *l*-element vector. *tau* and *z(k)* are chosen to annihilate the elements of the *k*-th row of *A*.

The scalar *tau* is returned in the *k*-th element of *tau* and the vector *u(k)* in the *k*-th row of *A*, such that the elements of *z(k)* are in *a(k, m+1), ..., a(k, n)*.

The elements of *R* are returned in the upper triangular part of *A*.

The matrix *Z* is given by

$$Z = Z(1) * Z(2) * \dots * Z(m).$$

For better performance, try using *lwork = m*blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork = -1*.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If *lwork = -1*, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Related routines include:

[ormrz](#) to apply matrix Q (for real matrices)

[unmrz](#) to apply matrix Q (for complex matrices).

?ormrz

Multiples a real matrix by the orthogonal matrix defined from the factorization formed by ?tzrzf.

Syntax

FORTRAN 77:

```
call sormrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call dormrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormrz(a, tau, c, l [, side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>ormrz( int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, const <datatype>* a, lapack_int lda,
const <datatype>* tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The ?ormrz routine multiplies a real m -by- n matrix C by Q or Q^T , where Q is the real orthogonal matrix defined as a product of k elementary reflectors $H(i)$ of order n : $Q = H(1) * H(2) * \dots * H(k)$ as returned by the factorization routine **tzrzf/tzrf**.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , Q^T*C , C^*Q , or C^*Q^T (overwriting the result over C).

The matrix Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

The ?ormrz routine replaces the deprecated ?latzm routine.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <i>side</i> = 'L'; $0 \leq k \leq n$, if <i>side</i> = 'R'.
<i>l</i>	INTEGER. The number of columns of the matrix A containing the meaningful part of the Householder reflectors. Constraints: $0 \leq l \leq m$, if <i>side</i> = 'L';

$0 \leq l \leq n$, if $\text{side} = 'R'$.

$a, tau, c, work$

REAL for sormrz

DOUBLE PRECISION for dormrz.

Arrays: $a(lda,*), tau(*), c(ldc,*)$.

On entry, the i th row of a must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by stzrzf/dtzrzf in the last k rows of its array argument a .

The second dimension of a must be at least $\max(1, m)$ if $\text{side} = 'L'$, and at least $\max(1, n)$ if $\text{side} = 'R'$.

$tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by stzrzf/dtzrzf.

The dimension of tau must be at least $\max(1, k)$.

$c(ldc,*)$ contains the m -by- n matrix C .

The second dimension of c must be at least $\max(1, n)$

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The leading dimension of a ; $lda \geq \max(1, k)$.

ldc

INTEGER. The leading dimension of c ; $ldc \geq \max(1, m)$.

$lwork$

INTEGER. The size of the $work$ array. Constraints:

$lwork \geq \max(1, n)$ if $\text{side} = 'L'$;

$lwork \geq \max(1, m)$ if $\text{side} = 'R'$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See [Application Notes](#) for the suggested value of $lwork$.

Output Parameters

c

Overwritten by the product Q^*C , Q^T*C , C^*Q , or C^*Q^T (as specified by side and trans).

$work(1)$

If $\text{info} = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$

INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormrz` interface are the following:

<code>a</code>	Holds the matrix A of size (k,m) .
<code>tau</code>	Holds the vector of length (k) .
<code>c</code>	Holds the matrix C of size (m,n) .
<code>side</code>	Must be ' <code>L</code> ' or ' <code>R</code> '. The default value is ' <code>L</code> '.
<code>trans</code>	Must be ' <code>N</code> ' or ' <code>T</code> '. The default value is ' <code>N</code> '.

Application Notes

For better performance, try using $lwork = n * \text{blocksize}$ (if `side = 'L'`) or $lwork = m * \text{blocksize}$ (if `side = 'R'`) where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex counterpart of this routine is [unmrz](#).

?unmrz

Multiplies a complex matrix by the unitary matrix defined from the factorization formed by ?tzrzf.

Syntax

FORTRAN 77:

```
call cunmrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
call zunmrz(side, trans, m, n, k, l, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmrz(a, tau, c, l [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>unmrz( int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, const <datatype>* a, lapack_int lda,
const <datatype>* tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine multiplies a complex m -by- n matrix C by Q or Q^H , where Q is the unitary matrix defined as a product of k elementary reflectors $H(i)$:

$Q = H(1)^H * H(2)^H * \dots * H(k)^H$ as returned by the factorization routine [tzrzf/tzrzf](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , $Q^{H*}C$, C^*Q , or C^*Q^H (overwriting the result over C).

The matrix Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^H is applied to C from the left. If <i>side</i> = 'R', Q or Q^H is applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'C'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'C', the routine multiplies C by Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: $0 \leq k \leq m$, if <i>side</i> = 'L'; $0 \leq k \leq n$, if <i>side</i> = 'R'.
<i>l</i>	INTEGER. The number of columns of the matrix A containing the meaningful part of the Householder reflectors. Constraints: $0 \leq l \leq m$, if <i>side</i> = 'L'; $0 \leq l \leq n$, if <i>side</i> = 'R'.
<i>a</i> , <i>tau</i> , <i>c</i> , <i>work</i>	COMPLEX for cunmrz DOUBLE COMPLEX for zunmrz . Arrays: <i>a</i> (<i>lda</i> ,*), <i>tau</i> (*), <i>c</i> (<i>ldc</i> ,*), <i>work</i> (<i>lwork</i>). On entry, the <i>i</i> th row of <i>a</i> must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ctzrzf/ztzrzf in the last k rows of its array argument <i>a</i> . The second dimension of <i>a</i> must be at least $\max(1, m)$ if <i>side</i> = 'L', and at least $\max(1, n)$ if <i>side</i> = 'R'. <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ctzrzf/ztzrzf .

The dimension of τ must be at least $\max(1, k)$.
 $c(ldc, *)$ contains the m -by- n matrix C .
The second dimension of c must be at least $\max(1, n)$
 $work$ is a workspace array, its dimension $\max(1, lwork)$.
 lda INTEGER. The leading dimension of a ; $lda \geq \max(1, k)$.
 ldc INTEGER. The leading dimension of c ; $ldc \geq \max(1, m)$.
 $lwork$ INTEGER. The size of the $work$ array. Constraints:
 $lwork \geq \max(1, n)$ if $side = 'L'$;
 $lwork \geq \max(1, m)$ if $side = 'R'$.
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by **xerbla**.
See *Application Notes* for the suggested value of $lwork$.

Output Parameters

c Overwritten by the product Q^*C , Q^H*C , C^*Q , or C^*Q^H (as specified by $side$ and $trans$).
 $work(1)$ If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
 $info$ INTEGER.
If $info = 0$, the execution is successful.
If $info = -i$, the i th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmrz` interface are the following:

a	Holds the matrix A of size (k,m) .
τ	Holds the vector of length (k) .
c	Holds the matrix C of size (m,n) .
$side$	Must be ' L ' or ' R '. The default value is ' L '.
$trans$	Must be ' N ' or ' C '. The default value is ' N '.

Application Notes

For better performance, try using $lwork = n*blocksize$ (if $side = 'L'$) or $lwork = m*blocksize$ (if $side = 'R'$) where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real counterpart of this routine is [ormrz](#).

?ggqrft

Computes the generalized QR factorization of two matrices.

Syntax

FORTRAN 77:

```
call sggqrft(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggqrft(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggqrft(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
call zggqrft(n, m, p, a, lda, taua, b, ldb, taub, work, lwork, info)
```

Fortran 95:

```
call ggqrft(a, b [,taua] [,taub] [,info])
```

C:

```
lapack_int LAPACKE_<?>ggqrft( int matrix_layout, lapack_int n, lapack_int m, lapack_int p, <datatype>* a, lapack_int lda, <datatype>* taua, <datatype>* b, lapack_int ldb, <datatype>* taub );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the generalized QR factorization of an *n*-by-*m* matrix *A* and an *n*-by-*p* matrix *B* as *A* = *Q***R*, *B* = *Q***T***Z*, where *Q* is an *n*-by-*n* orthogonal/unitary matrix, *Z* is a *p*-by-*p* orthogonal/unitary matrix, and *R* and *T* assume one of the forms:

$$R = \begin{cases} m & \begin{pmatrix} R_{11} \\ 0 \end{pmatrix}, \quad \text{if } n \geq m \\ n - m & \end{cases}$$

or

$$R = \begin{matrix} n & m - n \\ R_{11} & R_{12} \end{matrix}, \quad \text{if } n < m$$

where R_{11} is upper triangular, and

$$T = \begin{matrix} p - n & n \\ 0 & T_{12} \end{matrix}, \quad \text{if } n \leq p,$$

$$T = \begin{matrix} p \\ n - p \\ p \end{matrix} \begin{pmatrix} T_{11} \\ T_{12} \\ T_{21} \end{pmatrix}, \quad \text{if } n > p,$$

where T_{12} or T_{21} is a p -by- p upper triangular matrix.

In particular, if B is square and nonsingular, the GQR factorization of A and B implicitly gives the QR factorization of $B^{-1}A$ as:

$$B^{-1} * A = Z^{T*} (T^{-1} * R) \text{ (for real flavors) or } B^{-1} * A = Z^{H*} (T^{-1} * R) \text{ (for complex flavors).}$$

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The number of rows of the matrices A and B ($n \geq 0$).
<i>m</i>	INTEGER. The number of columns in A ($m \geq 0$).
<i>p</i>	INTEGER. The number of columns in B ($p \geq 0$).
<i>a, b, work</i>	REAL for <code>sggqr</code> DOUBLE PRECISION for <code>dggqr</code> COMPLEX for <code>cggqr</code> DOUBLE COMPLEX for <code>zggqr</code> . Arrays: $a(lda,*)$ contains the matrix A . The second dimension of a must be at least $\max(1, m)$. $b(lsb,*)$ contains the matrix B . The second dimension of b must be at least $\max(1, p)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of b ; at least $\max(1, n)$.

lwork INTEGER. The size of the *work* array; must be at least $\max(1, n, m, p)$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a, b

Overwritten by the factorization data as follows:

on exit, the elements on and above the diagonal of the array *a* contain the $\min(n,m)$ -by-*m* upper trapezoidal matrix *R* (*R* is upper triangular if $n \geq m$); the elements below the diagonal, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of $\min(n,m)$ elementary reflectors;

if $n \leq p$, the upper triangle of the subarray *b*(1:*n*, *p*-*n*+1:*p*) contains the *n*-by-*n* upper triangular matrix *T*;

if $n > p$, the elements on and above the (*n*-*p*)th subdiagonal contain the *n*-by-*p* upper trapezoidal matrix *T*; the remaining elements, with the array *taub*, represent the orthogonal/unitary matrix *Z* as a product of elementary reflectors.

taua, taub

REAL for sggqr

DOUBLE PRECISION for dggqr

COMPLEX for cggqr

DOUBLE COMPLEX for zggqr.

Arrays, DIMENSION at least max (1, $\min(n, m)$) for *taua* and at least max (1, $\min(n, p)$) for *taub*. The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*.

The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*.

work(1)

If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *ggqr* interface are the following:

a Holds the matrix *A* of size (*n,m*).

b Holds the matrix *B* of size (*n,p*).

*tau*a Holds the vector of length $\min(n,m)$.

taub Holds the vector of length $\min(n,p)$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(1)H(2)\dots H(k)$, where $k = \min(n, m)$.

Each $H(i)$ has the form

$H(i) = I - \tau_{ai} v v^T$ for real flavors, or

$H(i) = I - \tau \alpha u^* v^* v^H$ for complex flavors,

where τ_{au} is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$, $v(i) = 1$.

On exit, $v(i+1:n)$ is stored in $a(i+1:n, i)$ and τ_{au} is stored in $\tau_{au}(i)$.

The matrix Z is represented as a product of elementary reflectors

$Z = H(1)H(2)\dots H(k)$, where $k = \min(n, p)$.

Each $H(i)$ has the form

$H(i) = I - taub^* v^* v^T$ for real flavors, or

$H(i) = I - taub^* v^* v^H$ for complex flavors.

where $taub$ is a real/complex scalar, and v is a real/complex vector with $v(p-k+i+1:p) = 0$, $v(p-k+i) = 1$.

On exit, $y(1:p-k+i-1)$ is stored in $b(n-k+i, 1:p-k+i-1)$ and $taub$ is stored in $taub(i)$.

For better performance, try using `lwork = max(n, m, p) * max(nb1, nb2, nb3)`, where `nb1` is the optimal blocksize for the QR factorization of an n -by- m matrix, `nb2` is the optimal blocksize for the RQ factorization of an n -by- p matrix, and `nb3` is the optimal blocksize for a call of `ormqr/unmqr`.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggrgf

Computes the generalized RQ factorization of two matrices.

Syntax

FORTRAN 77:

```

call sggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call dggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call cggrqf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)
call zgaraf (m, p, n, a, lda, taua, b, ldb, taub, work, lwork, info)

```

Fortran 95:

```
call ggrqf(a, b [,taua] [,taub] [,info])
```

C:

```
lapack_int LAPACKE_<?>ggrqf( int matrix_layout, lapack_int m, lapack_int p, lapack_int n,
<datatype>* a, lapack_int lda, <datatype>* taua, <datatype>* b, lapack_int ldb,
<datatype>* taub );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the generalized *RQ* factorization of an m -by- n matrix A and an p -by- n matrix B as $A = R^*Q$, $B = Z^*T^*Q$, where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = \begin{matrix} n - m & m \\ m & \begin{pmatrix} 0 & R_{12} \end{pmatrix} \end{matrix}, \quad \text{if } m \leq n,$$

or

$$R = \begin{matrix} n \\ m - n \\ n \end{matrix} \begin{pmatrix} R_{11} \\ R_{21} \end{pmatrix}, \quad \text{if } m > n,$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{matrix} n \\ p - n \end{matrix} \begin{pmatrix} T_{11} \\ 0 \end{pmatrix}, \quad \text{if } p \geq n,$$

or

$$T = \begin{matrix} p & n - p \\ p & \begin{pmatrix} T_{11} & T_{12} \end{pmatrix} \end{matrix}, \quad \text{if } p < n,$$

where T_{11} is upper triangular.

In particular, if B is square and nonsingular, the *GRQ* factorization of A and B implicitly gives the *RQ* factorization of A^*B^{-1} as:

$A^*B^{-1} = (R^*T^{-1}) * Z^T$ (for real flavors) or $A^*B^{-1} = (R^*T^{-1}) * Z^H$ (for complex flavors).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>p</i>	INTEGER. The number of rows in B ($p \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrices A and B ($n \geq 0$).
<i>a, b, work</i>	REAL for sggqrqf DOUBLE PRECISION for dggrqf COMPLEX for cggrqf DOUBLE COMPLEX for zggrqf. Arrays: <i>a</i> (<i>lda,*</i>) contains the m -by- n matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb,*</i>) contains the p -by- n matrix B . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$. <i>lda</i> INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$. <i>ldb</i> INTEGER. The leading dimension of <i>b</i> ; at least $\max(1, p)$. <i>lwork</i> INTEGER. The size of the <i>work</i> array; must be at least $\max(1, n, m, p)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a, b</i>	Overwritten by the factorization data as follows: on exit, if $m \leq n$, the upper triangle of the subarray <i>a</i> (1: <i>m</i> , <i>n</i> - <i>m</i> +1: <i>n</i>) contains the m -by- m upper triangular matrix R ; if $m > n$, the elements on and above the $(m-n)$ th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array <i>taua</i> , represent the orthogonal/unitary matrix Q as a product of elementary reflectors; the elements on and above the diagonal of the array <i>b</i> contain the $\min(p, n)$ -by- n upper trapezoidal matrix T (T is upper triangular if $p \geq n$); the elements below the diagonal, with the array <i>taub</i> , represent the orthogonal/unitary matrix Z as a product of elementary reflectors.
-------------	---

<i>taua, taub</i>	REAL for sggrqf DOUBLE PRECISION for dggrqf COMPLEX for cggrqf DOUBLE COMPLEX for zggrqf. Arrays, DIMENSION at least max (1, min(m, n)) for <i>taua</i> and at least max (1, min(p, n)) for <i>taub</i> . The array <i>taua</i> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q . The array <i>taub</i> contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Z .
<i>work(1)</i>	If <i>info</i> = 0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *ggrqf* interface are the following:

<i>a</i>	Holds the matrix A of size (m, n).
<i>b</i>	Holds the matrix A of size (p, n).
<i>taua</i>	Holds the vector of length $\min(m, n)$.
<i>taub</i>	Holds the vector of length $\min(p, n)$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(1)H(2)\dots H(k), \text{ where } k = \min(m, n).$$

Each $H(i)$ has the form

$$H(i) = I - \tau_{aua} v v^T \text{ for real flavors, or}$$

$$H(i) = I - \tau_{aua} v v^H \text{ for complex flavors,}$$

where *taua* is a real/complex scalar, and *v* is a real/complex vector with $v(n-k+i+1:n) = 0$, $v(n-k+i) = 1$.

On exit, $v(1:n-k+i-1)$ is stored in $a(m-k+i, 1:n-k+i-1)$ and *taua* is stored in *taua(i)*.

The matrix Z is represented as a product of elementary reflectors

$$Z = H(1)H(2)\dots H(k), \text{ where } k = \min(p, n).$$

Each $H(i)$ has the form

$$H(i) = I - \tau_{aub} v v^T \text{ for real flavors, or}$$

$H(i) = I - taub * v * v^H$ for complex flavors,

where $taub$ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$, $v(i) = 1$.

On exit, $v(i+1:p)$ is stored in $b(i+1:p, i)$ and $taub$ is stored in $taub(i)$.

For better performance, try using

$lwork \geq \max(n, m, p) * \max(nb1, nb2, nb3)$,

where $nb1$ is the optimal blocksize for the RQ factorization of an m -by- n matrix, $nb2$ is the optimal blocksize for the QR factorization of an p -by- n matrix, and $nb3$ is the optimal blocksize for a call of $?ormrq/?unmrq$.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tpqrt

Computes a blocked QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for Q .

Syntax

FORTRAN 77:

```
call stpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
call dtpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
call ctpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
call ztpqrt(m, n, l, nb, a, lda, b, ldb, t, ldt, work, info)
```

Fortran 95:

```
call tpqrt(a, b, t, nb [, info])
```

C:

```
lapack_int LAPACKE_<?>tpqrt(int matrix_layout, lapack_int m, lapack_int n, lapack_int l, lapack_int nb, <datatype>* a, lapack_int lda, <datatype>* b, lapack_int ldb, <datatype>* t, lapack_int ldt)
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The input matrix C is an $(n+m)$ -by- n matrix

$$C = \begin{bmatrix} A \\ B \end{bmatrix} \begin{array}{l} \leftarrow n \times n \text{ upper triangular} \\ \leftarrow m \times n \text{ pentagonal} \end{array}$$

where A is an n -by- n upper triangular matrix, and B is an m -by- n pentagonal matrix consisting of an $(m-l)$ -by- n rectangular matrix $B1$ on top of an l -by- n upper trapezoidal matrix $B2$:

$$B = \begin{bmatrix} B1 \\ B2 \end{bmatrix} \begin{array}{l} \leftarrow (m-l) \times n \text{ rectangular} \\ \leftarrow l \times n \text{ upper trapezoidal} \end{array}$$

The upper trapezoidal matrix $B2$ consists of the first l rows of an n -by- n upper triangular matrix, where $0 \leq l \leq \min(m, n)$. If $l=0$, B is an m -by- n rectangular matrix. If $m=l=n$, B is upper triangular. The matrix W contains the elementary reflectors $H(i)$ in the i th column below the diagonal (of A) in the $(n+m)$ -by- n input matrix C so that W can be represented as

$$W = \begin{bmatrix} I \\ V \end{bmatrix} \begin{array}{l} \leftarrow n \times n \text{ identity} \\ \leftarrow m \times n \text{ pentagonal} \end{array}$$

Thus, V contains all of the information needed for W , and is returned in array b .

NOTE

Note that V has the same form as B :

$$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix} \begin{array}{l} \leftarrow (m-l) \times n \text{ rectangular} \\ \leftarrow l \times n \text{ upper trapezoidal} \end{array}$$

The columns of V represent the vectors which define the $H(i)$ s. The number of blocks is $k = \text{ceiling}(n/nb)$, where each block is of order nb except for the last block, which is of order $ib = n - (k-1)*nb$. For each of the k blocks, an upper triangular block reflector factor is computed: $T1, T2, \dots, Tb$. The nb -by- nb (ib -by- ib for the last block) T is are stored in the nb -by- n array t as

$$t = [T1 \ T2 \ \dots \ TB].$$

Input Parameters

m	INTEGER. The total number of rows in the matrix B ($m \geq 0$).
n	INTEGER. The number of columns in B and the order of the triangular matrix A ($n \geq 0$).
l	INTEGER. The number of rows of the upper trapezoidal part of B ($\min(m, n) \geq l \geq 0$).

<i>nb</i>	INTEGER. The block size to use in the blocked QR factorization ($n \geq nb \geq 1$).
<i>a, b, work</i>	<p>REAL for <code>stpmqrt</code></p> <p>DOUBLE PRECISION for <code>dtpmqrt</code></p> <p>COMPLEX for <code>ctpmqrt</code></p> <p>COMPLEX*16 for <code>ztpmqrt</code>.</p> <p>Arrays: <i>a</i> DIMENSION (<i>lda, n</i>) contains the <i>n</i>-by-<i>n</i> upper triangular matrix <i>A</i>.</p> <p><i>b</i> DIMENSION (<i>ldb, n</i>), the pentagonal <i>m</i>-by-<i>n</i> matrix <i>B</i>. The first (<i>m-l</i>) rows contain the rectangular <i>B1</i> matrix, and the next <i>l</i> rows contain the upper trapezoidal <i>B2</i> matrix.</p> <p><i>work</i> DIMENSION (<i>nb, n</i>) is a workspace array.</p>
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; at least $\max(1, m)$.
<i>ldt</i>	INTEGER. The leading dimension of <i>t</i> ; at least <i>nb</i> .

Output Parameters

<i>a</i>	The elements on and above the diagonal of the array contain the upper triangular matrix <i>R</i> .
<i>b</i>	The pentagonal matrix <i>V</i> .
<i>t</i>	<p>REAL for <code>stpmqrt</code></p> <p>DOUBLE PRECISION for <code>dtpmqrt</code></p> <p>COMPLEX for <code>ctpmqrt</code></p> <p>COMPLEX*16 for <code>ztpmqrt</code>.</p> <p>Array, DIMENSION (<i>ldt, n</i>).</p> <p>The upper triangular block reflectors stored in compact form as a sequence of upper triangular blocks.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> < 0 and <i>info</i> = -<i>i</i>, the <i>i</i>th argument had an illegal value.</p>

?tpmqrt

Applies a real or complex orthogonal matrix obtained from a "triangular-pentagonal" complex block reflector to a general real or complex matrix, which consists of two blocks.

Syntax

FORTRAN 77:

```
call stpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call dtpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
```

```
call ctpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
call ztpmqrt(side, trans, m, n, k, l, nb, v, ldv, t, ldt, a, lda, b, ldb, work, info)
```

Fortran 95:

```
call tpmqrt( v, t, a, b, k, nb [, trans] [, side] [, info])
```

C:

```
lapack_int LAPACKE_<?>tpmqrt(int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, lapack_int nb, const <datatype>* v,
lapack_int ldv, const <datatype>* t, lapack_int ldt, <datatype>* a, lapack_int lda,
<datatype>* b, lapack_int ldb)
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The columns of the pentagonal matrix V contain the elementary reflectors $H(1), H(2), \dots, H(k)$; V is composed of a rectangular block $V1$ and a trapezoidal block $V2$:

$$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix}$$

The size of the trapezoidal block $V2$ is determined by the parameter l , where $0 \leq l \leq k$. $V2$ is upper trapezoidal, consisting of the first l rows of a k -by- k upper triangular matrix.

If $l=k$, $V2$ is upper triangular;

If $l=0$, there is no trapezoidal block, so $V = V1$ is rectangular.

If $side = 'L'$:

$$C = \begin{bmatrix} A \\ B \end{bmatrix}$$

where A is k -by- n , B is m -by- n and V is m -by- k .

If $side = 'R'$:

$$C = [A \quad B]$$

where A is m -by- k , B is m -by- n and V is n -by- k .

The real/complex orthogonal matrix Q is formed from V and T .

If $trans='N'$ and $side='L'$, c contains $Q * C$ on exit.

If $trans='T'$ and $side='L'$, c contains $Q^T * C$ on exit.

If $trans='C'$ and $side='L'$, c contains $Q^H * C$ on exit.

If $\text{trans} = \text{'N'}$ and $\text{side} = \text{'R'}$, C contains $C * Q$ on exit.

If $\text{trans} = \text{'T'}$ and $\text{side} = \text{'R'}$, C contains $C * Q^T$ on exit.

If $\text{trans} = \text{'C'}$ and $\text{side} = \text{'R'}$, C contains $C * Q^H$ on exit.

Input Parameters

<i>side</i>	CHARACTER*1
	= 'L': apply Q , Q^T , or Q^H from the left.
	= 'R': apply Q , Q^T , or Q^H from the right.
<i>trans</i>	CHARACTER*1
	= 'N', no transpose, apply Q .
	= 'T', transpose, apply Q^T .
	= 'C', transpose, apply Q^H .
<i>m</i>	INTEGER. The number of rows in the matrix B , ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix B , ($n \geq 0$).
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q .
<i>l</i>	INTEGER. The order of the trapezoidal part of V ($k \geq l \geq 0$).
<i>nb</i>	INTEGER. The block size used for the storage of t , $k \geq nb \geq 1$. This must be the same value of nb used to generate t in tpqrt .
<i>v</i>	REAL for stpmqrt DOUBLE PRECISION for dtpmqrt COMPLEX for ctpmqrt COMPLEX*16 for ztpmqrt . DIMENSION (ldv, k) The i th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by tpqrt in array argument b .
<i>ldv</i>	INTEGER. The leading dimension of the array v . If $\text{side} = \text{'L'}$, ldv must be at least $\max(1, m)$; If $\text{side} = \text{'R'}$, ldv must be at least $\max(1, n)$.
<i>t</i>	REAL for stpmqrt DOUBLE PRECISION for dtpmqrt COMPLEX for ctpmqrt COMPLEX*16 for ztpmqrt . Array, DIMENSION (ldt, k).

The upper triangular factors of the block reflectors as returned by [tpqrt](#), stored as an nb -by- k matrix.

lbt INTEGER. The leading dimension of the array *t*. *lbt* must be at least *nb*.

a REAL for *stpmqrt*

DOUBLE PRECISION for *dtpmqrt*

COMPLEX for *ctpmqrt*

COMPLEX*16 for *ztpmqrt*.

If *side* = 'L', DIMENSION (*lda*, *n*).

If *side* = 'R', DIMENSION (*lda*, *k*).

The k -by- n or m -by- k matrix *A*.

lda INTEGER. The leading dimension of the array *a*.

If *side* = 'L', *lda* must be at least $\max(1, k)$.

If *side* = 'R', *lda* must be at least $\max(1, m)$.

b REAL for *stpmqrt*

DOUBLE PRECISION for *dtpmqrt*

COMPLEX for *ctpmqrt*

COMPLEX*16 for *ztpmqrt*.

DIMENSION (*ldb*, *n*).

The m -by- n matrix *B*.

ldb INTEGER. The leading dimension of the array *b*. *ldb* must be at least $\max(1, m)$.

work REAL for *stpmqrt*

DOUBLE PRECISION for *dtpmqrt*

COMPLEX for *ctpmqrt*

COMPLEX*16 for *ztpmqrt*.

Workspace array. If *side* = 'L' DIMENSION $n * nb$. If *side* = 'R' DIMENSION $m * nb$.

Output Parameters

a Overwritten by the corresponding block of the product Q^*C , C^*Q , Q^T*C , C^*Q^T , $Q^{H*}C$, or C^*Q^H .

b Overwritten by the corresponding block of the product Q^*C , C^*Q , Q^T*C , C^*Q^T , $Q^{H*}C$, or C^*Q^H .

info (global) INTEGER.

= 0: the execution is successful.

< 0: if *info* = $-i$, the *i*th argument had an illegal value.

Singular Value Decomposition - LAPACK Computational Routines

This section describes LAPACK routines for computing the *singular value decomposition* (SVD) of a general m -by- n matrix A :

$$A = U\Sigma V^H.$$

In this decomposition, U and V are unitary (for complex A) or orthogonal (for real A); Σ is an m -by- n diagonal matrix with real diagonal elements σ_i :

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\min(m, n)} \geq 0.$$

The diagonal elements σ_i are *singular values* of A . The first $\min(m, n)$ columns of the matrices U and V are, respectively, *left* and *right singular vectors* of A . The singular values and singular vectors satisfy

$$Av_i = \sigma_i u_i \text{ and } A^H v_i = \sigma_i v_i$$

where u_i and v_i are the i -th columns of U and V , respectively.

To find the SVD of a general matrix A , call the LAPACK routine `?gebrd` or `?gbbrd` for reducing A to a bidiagonal matrix B by a unitary (orthogonal) transformation: $A = QBP^H$. Then call `?bdsqr`, which forms the SVD of a bidiagonal matrix: $B = U_1 \Sigma V_1^H$.

Thus, the sought-for SVD of A is given by $A = U\Sigma V^H = (QU_1)\Sigma(V_1^H P^H)$.

Table "Computational Routines for Singular Value Decomposition (SVD)" lists LAPACK routines (FORTRAN 77 interface) that perform singular value decomposition of matrices. The corresponding routine names in the Fortran 95 interface are the same except that the first character is removed.

Computational Routines for Singular Value Decomposition (SVD)

Operation	Real matrices	Complex matrices
Reduce A to a bidiagonal matrix B : $A = QBP^H$ (full storage)	<code>?gebrd</code>	<code>?gebrd</code>
Reduce A to a bidiagonal matrix B : $A = QBP^H$ (band storage)	<code>?gbbrd</code>	<code>?gbbrd</code>
Generate the orthogonal (unitary) matrix Q or P	<code>?orgbr</code>	<code>?ungbr</code>
Apply the orthogonal (unitary) matrix Q or P	<code>?ormbr</code>	<code>?unmbr</code>
Form singular value decomposition of the bidiagonal matrix B : $B = U\Sigma V^H$	<code>?bdsqr</code> <code>?bdsdc</code>	<code>?bdsqr</code>

Decision Tree: Singular Value Decomposition

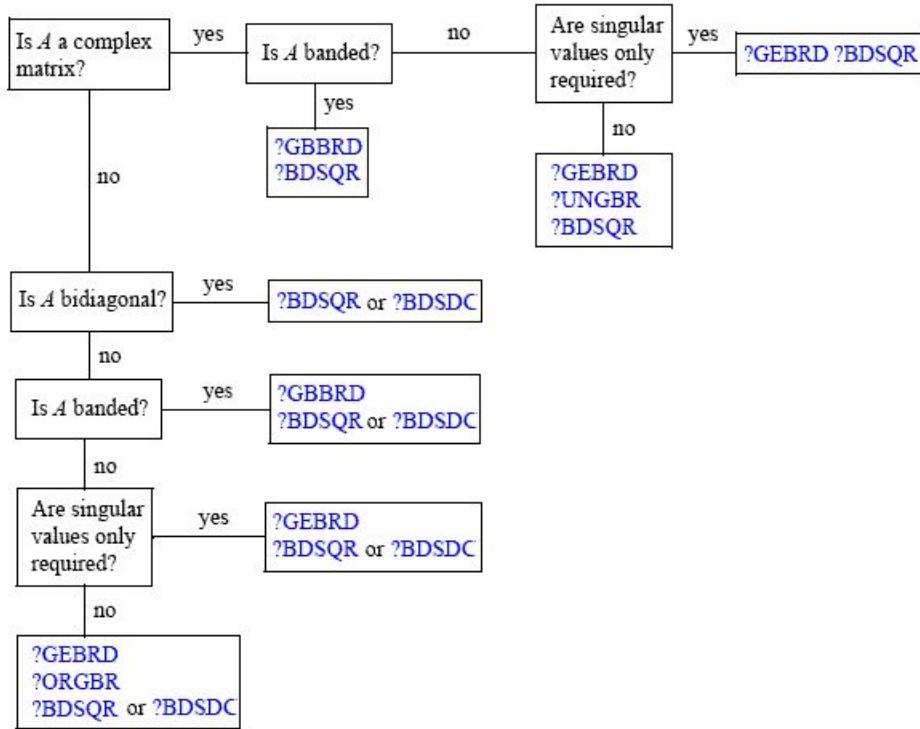


Figure "Decision Tree: Singular Value Decomposition" presents a decision tree that helps you choose the right sequence of routines for SVD, depending on whether you need singular values only or singular vectors as well, whether A is real or complex, and so on.

You can use the SVD to find a minimum-norm solution to a (possibly) rank-deficient least squares problem of minimizing $\|Ax - b\|^2$. The effective rank k of the matrix A can be determined as the number of singular values which exceed a suitable threshold. The minimum-norm solution is

$$x = V_k(\Sigma_k)^{-1}c$$

where Σ_k is the leading k -by- k submatrix of Σ , the matrix V_k consists of the first k columns of $V = PV_1$, and the vector c consists of the first k elements of $U^Hb = U_1^HQ^Hb$.

?gebrd

Reduces a general matrix to bidiagonal form.

Syntax

FORTRAN 77:

```

call sgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call dgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call cgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
call zgebrd(m, n, a, lda, d, e, tauq, taup, work, lwork, info)
  
```

Fortran 95:

```
call gebrd(a [, d] [, e] [,tauq] [,taup] [,info])
```

C:

```
lapack_int LAPACKE_sgebrd( int matrix_layout, lapack_int m, lapack_int n, float* a,
                           lapack_int lda, float* d, float* e, float* tauq, float* taup );
```

```

lapack_int LAPACKE_dgebrd( int matrix_layout, lapack_int m, lapack_int n, double* a,
                           lapack_int lda, double* d, double* e, double* tauq, double* taup );
lapack_int LAPACKE_cgebrd( int matrix_layout, lapack_int m, lapack_int n,
                           lapack_complex_float* a, lapack_int lda, float* d, float* e, lapack_complex_float* tauq,
                           lapack_complex_float* taup );
lapack_int LAPACKE_zgebrd( int matrix_layout, lapack_int m, lapack_int n,
                           lapack_complex_double* a, lapack_int lda, double* d, double* e, lapack_complex_double* tauq,
                           lapack_complex_double* taup );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces a general m -by- n matrix A to a bidiagonal matrix B by an orthogonal (unitary) transformation.

$$A = QBP^H = Q \begin{pmatrix} B_1 \\ 0 \end{pmatrix} P^H = Q_1 B_1 P^H,$$

If $m \geq n$, the reduction is given by

where B_1 is an n -by- n upper diagonal matrix, Q and P are orthogonal or, for a complex A , unitary matrices; Q_1 consists of the first n columns of Q .

If $m < n$, the reduction is given by

$$A = Q^* B^* P^H = Q^* (B_1 0)^* P^H = Q_1^* B_1^* P_1^H,$$

where B_1 is an m -by- m lower diagonal matrix, Q and P are orthogonal or, for a complex A , unitary matrices; P_1 consists of the first m rows of P .

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices Q and P in this representation:

If the matrix A is real,

- to compute Q and P explicitly, call [orgbr](#).
- to multiply a general matrix by Q or P , call [ormbr](#).

If the matrix A is complex,

- to compute Q and P explicitly, call [ungbr](#).
- to multiply a general matrix by Q or P , call [unmbr](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a, work REAL for sgebrd

DOUBLE PRECISION for dgebrd

COMPLEX for cgebrd
DOUBLE COMPLEX for zgebrd.

Arrays:

a(*lda*,*) contains the matrix *A*.

The second dimension of *a* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

lwork INTEGER.

The dimension of *work*; at least $\max(1, m, n)$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a If $m \geq n$, the diagonal and first super-diagonal of *a* are overwritten by the upper bidiagonal matrix *B*. Elements below the diagonal are overwritten by details of *Q*, and the remaining elements are overwritten by details of *P*.

If $m < n$, the diagonal and first sub-diagonal of *a* are overwritten by the lower bidiagonal matrix *B*. Elements above the diagonal are overwritten by details of *P*, and the remaining elements are overwritten by details of *Q*.

d REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, DIMENSION at least $\max(1, \min(m, n))$.

Contains the diagonal elements of *B*.

e REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, DIMENSION at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of *B*.

taug, *taup* REAL for sgebrd

DOUBLE PRECISION for dgebrd

COMPLEX for cgebrd

DOUBLE COMPLEX for zgebrd.

Arrays, DIMENSION at least $\max(1, \min(m, n))$. Contain further details of the matrices *Q* and *P*.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.

If *info* = 0, the execution is successful.If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gebrd` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m,n</i>).
<i>d</i>	Holds the vector of length min(<i>m,n</i>).
<i>e</i>	Holds the vector of length min(<i>m,n</i>)-1.
<i>taug</i>	Holds the vector of length min(<i>m,n</i>).
<i>taup</i>	Holds the vector of length min(<i>m,n</i>).

Application Notes

For better performance, try using *lwork* = (*m* + *n*) * *blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrices *Q*, *B*, and *P* satisfy $QBP^H = A + E$, where $\|E\|_2 = c(n)\epsilon \|A\|_2$, $c(n)$ is a modestly increasing function of *n*, and ϵ is the machine precision.

The approximate number of floating-point operations for real flavors is

$$(4/3) * n^2 * (3*m - n) \text{ for } m \geq n,$$

$$(4/3) * m^2 * (3*n - m) \text{ for } m < n.$$

The number of operations for complex flavors is four times greater.

If *n* is much less than *m*, it can be more efficient to first form the QR factorization of *A* by calling `geqr` and then reduce the factor *R* to bidiagonal form. This requires approximately $2*n^2*(m + n)$ floating-point operations.

If *m* is much less than *n*, it can be more efficient to first form the LQ factorization of *A* by calling `gelq` and then reduce the factor *L* to bidiagonal form. This requires approximately $2*m^2*(m + n)$ floating-point operations.

?gbbrd

Reduces a general band matrix to bidiagonal form.

Syntax**FORTRAN 77:**

```
call sgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work,
info)

call dgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work,
info)

call cgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work,
rwork, info)

call zgbbrd(vect, m, n, ncc, kl, ku, ab, ldab, d, e, q, ldq, pt, ldpt, c, ldc, work,
rwork, info)
```

Fortran 95:

```
call gbbnd(ab [, c] [, d] [, e] [, q] [, pt] [, kl] [, m] [, info])
```

C:

```
lapack_int LAPACKE_sgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int ncc, lapack_int kl, lapack_int ku, float* ab, lapack_int ldab, float* d,
float* e, float* q, lapack_int ldq, float* pt, lapack_int ldpt, float* c, lapack_int ldc );

lapack_int LAPACKE_dgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int ncc, lapack_int kl, lapack_int ku, double* ab, lapack_int ldab, double* d,
double* e, double* q, lapack_int ldq, double* pt, lapack_int ldpt, double* c,
lapack_int ldc );

lapack_int LAPACKE_cgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int ncc, lapack_int kl, lapack_int ku, lapack_complex_float* ab, lapack_int ldab,
float* d, float* e, lapack_complex_float* q, lapack_int ldq,
lapack_complex_float* pt, lapack_int ldpt, lapack_complex_float* c, lapack_int ldc );

lapack_int LAPACKE_zgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int ncc, lapack_int kl, lapack_int ku, lapack_complex_double* ab, lapack_int ldab,
double* d, double* e, lapack_complex_double* q, lapack_int ldq,
lapack_complex_double* pt, lapack_int ldpt, lapack_complex_double* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces an m -by- n band matrix A to upper bidiagonal matrix B : $A = Q^H B P^H$. Here the matrices Q and P are orthogonal (for real A) or unitary (for complex A). They are determined as products of Givens rotation matrices, and may be formed explicitly by the routine if required. The routine can also update a matrix C as follows: $C = Q^H C$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>vect</i>	CHARACTER*1. Must be 'N' or 'Q' or 'P' or 'B'. If <i>vect</i> = 'N', neither <i>Q</i> nor P^H is generated. If <i>vect</i> = 'Q', the routine generates the matrix <i>Q</i> . If <i>vect</i> = 'P', the routine generates the matrix P^H . If <i>vect</i> = 'B', the routine generates both <i>Q</i> and P^H .
<i>m</i>	INTEGER. The number of rows in the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>ncc</i>	INTEGER. The number of columns in <i>C</i> ($ncc \geq 0$).
<i>k1</i>	INTEGER. The number of sub-diagonals within the band of <i>A</i> ($k1 \geq 0$).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of <i>A</i> ($ku \geq 0$).
<i>ab, c, work</i>	REAL for sgbbrd DOUBLE PRECISION for dgbbrd COMPLEX for cgbbrd DOUBLE COMPLEX for zgbbrd. Arrays: <i>ab</i> (<i>ldab</i> ,*) contains the matrix <i>A</i> in band storage (see Matrix Storage Schemes). The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>c</i> (<i>ldc</i> ,*) contains an <i>m</i> -by- <i>ncc</i> matrix <i>C</i> . If <i>ncc</i> = 0, the array <i>c</i> is not referenced. The second dimension of <i>c</i> must be at least $\max(1, ncc)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $2 * \max(m, n)$ for real flavors, or $\max(m, n)$ for complex flavors.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ($ldab \geq k1 + ku + 1$).
<i>ldq</i>	INTEGER. The leading dimension of the output array <i>q</i> . $ldq \geq \max(1, m)$ if <i>vect</i> = 'Q' or 'B', $ldq \geq 1$ otherwise.
<i>ldpt</i>	INTEGER. The leading dimension of the output array <i>pt</i> . $ldpt \geq \max(1, n)$ if <i>vect</i> = 'P' or 'B', $ldpt \geq 1$ otherwise.
<i>ldc</i>	INTEGER. The leading dimension of the array <i>c</i> . $ldc \geq \max(1, m)$ if <i>ncc</i> > 0; $ldc \geq 1$ if <i>ncc</i> = 0.
<i>rwork</i>	REAL for cgbbrd DOUBLE PRECISION for zgbbrd. A workspace array, DIMENSION at least $\max(m, n)$.

Output Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>ab</i>	Overwritten by values generated during the reduction.
<i>d</i>	<code>REAL</code> for single-precision flavors <code>DOUBLE PRECISION</code> for double-precision flavors. <code>Array, DIMENSION at least max(1, min(<i>m, n</i>)).</code> Contains the diagonal elements of the matrix <i>B</i> .
<i>e</i>	<code>REAL</code> for single-precision flavors <code>DOUBLE PRECISION</code> for double-precision flavors. <code>Array, DIMENSION at least max(1, min(<i>m, n</i>) - 1).</code> Contains the off-diagonal elements of <i>B</i> .
<i>q, pt</i>	<code>REAL</code> for <code>sgebrd</code> <code>DOUBLE PRECISION</code> for <code>dgebrd</code> <code>COMPLEX</code> for <code>cgebrd</code> <code>DOUBLE COMPLEX</code> for <code>zgebrd</code> . Arrays: <i>q</i> (<i>ldq, *</i>) contains the output <i>m</i> -by- <i>m</i> matrix <i>Q</i> . The second dimension of <i>q</i> must be at least <code>max(1, m)</code> . <i>p</i> (<i>ldpt, *</i>) contains the output <i>n</i> -by- <i>n</i> matrix <i>P</i> ^T . The second dimension of <i>pt</i> must be at least <code>max(1, n)</code> .
<i>c</i>	Overwritten by the product $Q^H * C$. <i>c</i> is not referenced if <i>ncc</i> = 0.
<i>info</i>	<code>INTEGER</code> . If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gbbrd` interface are the following:

<i>ab</i>	Holds the array <i>A</i> of size (<i>k</i> <i>l</i> + <i>ku</i> +1, <i>n</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m, ncc</i>).
<i>d</i>	Holds the vector with the number of elements <code>min(m,n)</code> .

<i>e</i>	Holds the vector with the number of elements $\min(m,n)-1$.
<i>q</i>	Holds the matrix Q of size (m,m) .
<i>pt</i>	Holds the matrix PT of size (n,n) .
<i>m</i>	If omitted, assumed $m = n$.
<i>k1</i>	If omitted, assumed $k1 = ku$.
<i>ku</i>	Restored as $ku = lda - k1 - 1$.
<i>vect</i>	Restored based on the presence of arguments <i>q</i> and <i>pt</i> as follows: <i>vect</i> = 'B', if both <i>q</i> and <i>pt</i> are present, <i>vect</i> = 'Q', if <i>q</i> is present and <i>pt</i> omitted, <i>vect</i> = 'P', if <i>q</i> is omitted and <i>pt</i> present, <i>vect</i> = 'N', if both <i>q</i> and <i>pt</i> are omitted.

Application Notes

The computed matrices Q , B , and P satisfy $Q^*B^*P^H = A + E$, where $\|E\|_2 = c(n)\varepsilon \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ε is the machine precision.

If $m = n$, the total number of floating-point operations for real flavors is approximately the sum of:

$6*n^2*(k1 + ku)$ if *vect* = 'N' and *ncc* = 0,

$3*n^2*ncc*(k1 + ku - 1)/(k1 + ku)$ if *C* is updated, and

$3*n^3*(k1 + ku - 1)/(k1 + ku)$ if either *Q* or P^H is generated (double this if both).

To estimate the number of operations for complex flavors, use the same formulas with the coefficients 20 and 10 (instead of 6 and 3).

?orgbr

Generates the real orthogonal matrix Q or PT determined by ?gebrd.

Syntax

FORTRAN 77:

```
call sorgbr(vect, m, n, k, a, lda, tau, work, lwork, info)
call dorgbr(vect, m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgbr(a, tau [,vect] [,info])
```

C:

```
lapack_int LAPACKE_<?>orgbr( int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int k, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine generates the whole or part of the orthogonal matrices Q and P^T formed by the routines [gebrd](#)/[gebrd](#). Use this routine after a call to [sgebrd](#)/[dgebrd](#). All valid combinations of arguments are described in *Input parameters*. In most cases you need the following:

To compute the whole m -by- m matrix Q :

```
call ?orgbr('Q', m, m, n, a ... )
```

(note that the array a must have at least m columns).

To form the n leading columns of Q if $m > n$:

```
call ?orgbr('Q', m, n, n, a ... )
```

To compute the whole n -by- n matrix P^T :

```
call ?orgbr('P', n, n, m, a ... )
```

(note that the array a must have at least n rows).

To form the m leading rows of P^T if $m < n$:

```
call ?orgbr('P', m, n, m, a ... )
```

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>vect</i>	CHARACTER*1. Must be ' <i>Q</i> ' or ' <i>P</i> '.
	If <i>vect</i> = ' <i>Q</i> ', the routine generates the matrix Q .
	If <i>vect</i> = ' <i>P</i> ', the routine generates the matrix P^T .
<i>m, n</i>	INTEGER. The number of rows (m) and columns (n) in the matrix Q or P^T to be returned ($m \geq 0, n \geq 0$).
	If <i>vect</i> = ' <i>Q</i> ', $m \geq n \geq \min(m, k)$.
	If <i>vect</i> = ' <i>P</i> ', $n \geq m \geq \min(n, k)$.
<i>k</i>	If <i>vect</i> = ' <i>Q</i> ', the number of columns in the original m -by- k matrix reduced by gebrd .
	If <i>vect</i> = ' <i>P</i> ', the number of rows in the original k -by- n matrix reduced by gebrd .
<i>a</i>	REAL for sorgbr DOUBLE PRECISION for dorgbr
	The vectors which define the elementary reflectors, as returned by gebrd .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $\text{lda} \geq \max(1, m)$.
<i>tau</i>	REAL for sorgbr DOUBLE PRECISION for dorgbr

Array, DIMENSION $\min(m, k)$ if *vect* = 'Q', $\min(n, k)$ if *vect* = 'P'. Scalar factor of the elementary reflector $H(i)$ or $G(i)$, which determines Q and P^T as returned by **gebrd** in the array *tauq* or *taup*.

work REAL **for** *sorgbr*

DOUBLE PRECISION **for** *dorgbr*

Workspace array, DIMENSION $\max(1, lwork)$.

lwork INTEGER. Dimension of the array *work*. See *Application Notes* for the suggested value of *lwork*.

If *lwork* = -1 then the routine performs a workspace query and calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by **xerbla**.

Output Parameters

a Overwritten by the orthogonal matrix Q or P^T (or the leading rows or columns thereof) as specified by *vect*, *m*, and *n*.

work(1) If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *orgbr* interface are the following:

a Holds the matrix A of size (m,n) .

tau Holds the vector of length $\min(m,k)$ where

$k = m$, if *vect* = 'P',

$k = n$, if *vect* = 'Q'.

vect Must be 'Q' or 'P'. The default value is 'Q'.

Application Notes

For better performance, try using *lwork* = $\min(m, n) * \text{blocksize}$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the blocked algorithm.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)$.

The approximate numbers of floating-point operations for the cases listed in *Description* are as follows:

To form the whole of Q :

$(4/3) * n * (3m^2 - 3m*n + n^2)$ if $m > n$;

$(4/3) * m^3$ if $m \leq n$.

To form the n leading columns of Q when $m > n$:

$(2/3) * n^2 * (3m - n^2)$ if $m > n$.

To form the whole of P^T :

$(4/3) * n^3$ if $m \geq n$;

$(4/3) * m * (3n^2 - 3m*n + m^2)$ if $m < n$.

To form the m leading columns of P^T when $m < n$:

$(2/3) * n^2 * (3m - n^2)$ if $m > n$.

The complex counterpart of this routine is [ungbr](#).

?ormbr

Multiples an arbitrary real matrix by the real orthogonal matrix Q or P^T determined by ?gebrd.

Syntax

FORTRAN 77:

```
call sormbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call dormbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormbr(a, tau, c [,vect] [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>ormbr( int matrix_layout, char vect, char side, char trans,
lapack_int m, lapack_int n, lapack_int k, const <datatype>* a, lapack_int lda,
const <datatype>* tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

Given an arbitrary real matrix C , this routine forms one of the matrix products Q^*C , Q^TC , C^*Q , C^*Q^T , P^*C , P^TC , C^*P , C^*P^T , where Q and P are orthogonal matrices computed by a call to [gebrd/gebrd](#). The routine overwrites the product on C .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

In the descriptions below, r denotes the order of Q or P^T :

If $\text{side} = \text{'L'}$, $r = m$; if $\text{side} = \text{'R'}$, $r = n$.

<i>vect</i>	CHARACTER*1. Must be ' Q ' or ' P '. If $\text{vect} = \text{'Q'}$, then Q or Q^T is applied to C . If $\text{vect} = \text{'P'}$, then P or P^T is applied to C .
<i>side</i>	CHARACTER*1. Must be ' L ' or ' R '. If $\text{side} = \text{'L'}$, multipliers are applied to C from the left. If $\text{side} = \text{'R'}$, they are applied to C from the right.
<i>trans</i>	CHARACTER*1. Must be ' N ' or ' T '. If $\text{trans} = \text{'N'}$, then Q or P is applied to C . If $\text{trans} = \text{'T'}$, then Q^T or P^T is applied to C .
<i>m</i>	INTEGER. The number of rows in C .
<i>n</i>	INTEGER. The number of columns in C .
<i>k</i>	INTEGER. One of the dimensions of A in ?gebrd: If $\text{vect} = \text{'Q'}$, the number of columns in A ; If $\text{vect} = \text{'P'}$, the number of rows in A . Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.
<i>a, c, work</i>	REAL for sormbr DOUBLE PRECISION for dormbr. Arrays: $a(\text{lda},*)$ is the array a as returned by ?gebrd. Its second dimension must be at least $\max(1, \min(r, k))$ for $\text{vect} = \text{'Q'}$, or $\max(1, r)$ for $\text{vect} = \text{'P'}$. $c(\text{ldc},*)$ holds the matrix C . Its second dimension must be at least $\max(1, n)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of a . Constraints: $\text{lda} \geq \max(1, r)$ if $\text{vect} = \text{'Q'}$; $\text{lda} \geq \max(1, \min(r, k))$ if $\text{vect} = \text{'P'}$.
<i>ldc</i>	INTEGER. The leading dimension of c ; $\text{ldc} \geq \max(1, m)$.
<i>tau</i>	REAL for sormbr

DOUBLE PRECISION for `dormbr`.
 Array, DIMENSION at least max (1, min(r , k)).
 For `vect = 'Q'`, the array `tauq` as returned by `?gebrd`. For `vect = 'P'`, the array `taup` as returned by `?gebrd`.

`lwork` INTEGER. The size of the `work` array. Constraints:

$lwork \geq \max(1, n)$ if `side = 'L'`;
 $lwork \geq \max(1, m)$ if `side = 'R'`.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`.

See *Application Notes* for the suggested value of `lwork`.

Output Parameters

`c` Overwritten by the product Q^*C , Q^T*C , C^*Q , C^*Q^T , P^*C , P^T*C , C^*P , or C^*P^T , as specified by `vect`, `side`, and `trans`.

`work(1)` If `info = 0`, on exit `work(1)` contains the minimum value of `lwork` required for optimum performance. Use this `lwork` for subsequent runs.

`info` INTEGER.
 If `info = 0`, the execution is successful.
 If `info = -i`, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormbr` interface are the following:

`a` Holds the matrix A of size $(r, \min(nq, k))$ where
 $r = nq$, if `vect = 'Q'`,
 $r = \min(nq, k)$, if `vect = 'P'`,
 $nq = m$, if `side = 'L'`,
 $nq = n$, if `side = 'R'`,
 $k = m$, if `vect = 'P'`,
 $k = n$, if `vect = 'Q'`.

`tau` Holds the vector of length $\min(nq, k)$.

`c` Holds the matrix C of size (m, n) .

`vect` Must be '`Q`' or '`P`'. The default value is '`Q`'.

`side` Must be '`L`' or '`R`'. The default value is '`L`'.

trans Must be '`N`' or '`T`'. The default value is '`N`'.

Application Notes

For better performance, try using

```
lwork = n*blocksize for side = 'L', or
lwork = m*blocksize for side = 'R',
```

where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\epsilon) * \|C\|_2$.

The total number of floating-point operations is approximately

```
2*n*k(2*m - k) if side = 'L' and m ≥ k;
2*m*k(2*n - k) if side = 'R' and n ≥ k;
2*m^2*n if side = 'L' and m < k;
2*n^2*m if side = 'R' and n < k.
```

The complex counterpart of this routine is [unmbr](#).

?ungbr

Generates the complex unitary matrix *Q* or *P^H* determined by ?gebrd.

Syntax

FORTRAN 77:

```
call cungbr(vect, m, n, k, a, lda, tau, work, lwork, info)
call zungbr(vect, m, n, k, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungbr(a, tau [,vect] [,info])
```

C:

```
lapack_int LAPACKE_<?>ungbr( int matrix_layout, char vect, lapack_int m, lapack_int n,
lapack_int k, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`

- C: mkl.h

Description

The routine generates the whole or part of the unitary matrices Q and P^H formed by the routines [gebrd](#)/[gebrd](#). Use this routine after a call to [cgebrd](#)/[zgebrd](#). All valid combinations of arguments are described in *Input Parameters*; in most cases you need the following:

To compute the whole m -by- m matrix Q , use:

```
call ?ungbr('Q', m, m, n, a ... )
```

(note that the array a must have at least m columns).

To form the n leading columns of Q if $m > n$, use:

```
call ?ungbr('Q', m, n, n, a ... )
```

To compute the whole n -by- n matrix P^H , use:

```
call ?ungbr('P', n, n, m, a ... )
```

(note that the array a must have at least n rows).

To form the m leading rows of P^H if $m < n$, use:

```
call ?ungbr('P', m, n, m, a ... )
```

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>vect</i>	CHARACTER*1. Must be ' <i>Q</i> ' or ' <i>P</i> '.
	If <i>vect</i> = ' <i>Q</i> ', the routine generates the matrix Q .
	If <i>vect</i> = ' <i>P</i> ', the routine generates the matrix P^H .
<i>m</i>	INTEGER. The number of required rows of Q or P^H .
<i>n</i>	INTEGER. The number of required columns of Q or P^H .
<i>k</i>	INTEGER. One of the dimensions of A in ?gebrd :
	If <i>vect</i> = ' <i>Q</i> ', the number of columns in A ;
	If <i>vect</i> = ' <i>P</i> ', the number of rows in A .
	Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.
	For <i>vect</i> = ' <i>Q</i> ': $k \leq n \leq m$ if $m > k$, or $m = n$ if $m \leq k$.
	For <i>vect</i> = ' <i>P</i> ': $k \leq m \leq n$ if $n > k$, or $m = n$ if $n \leq k$.
<i>a, work</i>	COMPLEX for cungbr DOUBLE COMPLEX for zungbr .
	Arrays: <i>a</i> (<i>lda</i> ,*) is the array a as returned by ?gebrd . The second dimension of a must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, \text{lwork})$.

<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>tau</i>	COMPLEX for <i>cungbr</i> DOUBLE COMPLEX for <i>zungbr</i> . For <i>vect</i> = 'Q', the array <i>tauq</i> as returned by <i>?gebrd</i> . For <i>vect</i> = 'P', the array <i>taup</i> as returned by <i>?gebrd</i> . The dimension of <i>tau</i> must be at least $\max(1, \min(m, k))$ for <i>vect</i> = 'Q', or $\max(1, \min(m, k))$ for <i>vect</i> = 'P'.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraint: <i>lwork</i> < $\max(1, \min(m, n))$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the orthogonal matrix <i>Q</i> or <i>P</i> ^T (or the leading rows or columns thereof) as specified by <i>vect</i> , <i>m</i> , and <i>n</i> .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *ungbr* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>m,n</i>).
<i>tau</i>	Holds the vector of length $\min(m, k)$ where <i>k</i> = <i>m</i> , if <i>vect</i> = 'P', <i>k</i> = <i>n</i> , if <i>vect</i> = 'Q'.
<i>vect</i>	Must be 'Q' or 'P'. The default value is 'Q'.

Application Notes

For better performance, try using *lwork* = $\min(m, n) * \text{blocksize}$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from an exactly orthogonal matrix by a matrix *E* such that $\|E\|_2 = O(\varepsilon)$.

The approximate numbers of possible floating-point operations are listed below:

To compute the whole matrix *Q*:

$(16/3)n(3m^2 - 3m*n + n^2)$ if $m > n$;

$(16/3)m^3$ if $m \leq n$.

To form the *n* leading columns of *Q* when $m > n$:

$(8/3)n^2(3m - n^2)$.

To compute the whole matrix *P*^H:

$(16/3)n^3$ if $m \geq n$;

$(16/3)m(3n^2 - 3m*n + m^2)$ if $m < n$.

To form the *m* leading columns of *P*^H when $m < n$:

$(8/3)n^2(3m - n^2)$ if $m > n$.

The real counterpart of this routine is [orgbr](#).

?unmbr

Multiples an arbitrary complex matrix by the unitary matrix Q or P determined by ?gebrd.

Syntax

FORTRAN 77:

```
call cunmbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
call zunmbr(vect, side, trans, m, n, k, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmbr(a, tau, c [,vect] [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>unmbr( int matrix_layout, char vect, char side, char trans,
lapack_int m, lapack_int n, lapack_int k, const <datatype>* a, lapack_int lda,
const <datatype>* tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

Given an arbitrary complex matrix C , this routine forms one of the matrix products Q^*C , Q^H*C , C^*Q , C^*Q^H , P^*C , P^H*C , C^*P , or C^*P^H , where Q and P are unitary matrices computed by a call to [gebrd/gebrd](#). The routine overwrites the product on C .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

In the descriptions below, r denotes the order of Q or P^H :

If $\text{side} = \text{'L'}$, $r = m$; if $\text{side} = \text{'R'}$, $r = n$.

vect CHARACTER*1. Must be ' Q ' or ' P '.
 If $\text{vect} = \text{'Q'}$, then Q or Q^H is applied to C .
 If $\text{vect} = \text{'P'}$, then P or P^H is applied to C .

side CHARACTER*1. Must be ' L ' or ' R '.
 If $\text{side} = \text{'L'}$, multipliers are applied to C from the left.
 If $\text{side} = \text{'R'}$, they are applied to C from the right.

trans CHARACTER*1. Must be ' N ' or ' C '.
 If $\text{trans} = \text{'N'}$, then Q or P is applied to C .
 If $\text{trans} = \text{'C'}$, then Q^H or P^H is applied to C .

m INTEGER. The number of rows in C .

n INTEGER. The number of columns in C .

k INTEGER. One of the dimensions of A in ?gebrd:
 If $\text{vect} = \text{'Q'}$, the number of columns in A ;
 If $\text{vect} = \text{'P'}$, the number of rows in A .
 Constraints: $m \geq 0$, $n \geq 0$, $k \geq 0$.

a, c, work COMPLEX for cunmbr
 DOUBLE COMPLEX for zunmbr.

Arrays:

$a(lda,*)$ is the array a as returned by ?gebrd.

Its second dimension must be at least $\max(1, \min(r, k))$ for $\text{vect} = \text{'Q'}$, or $\max(1, r)$ for $\text{vect} = \text{'P'}$.

$c(ldc,*)$ holds the matrix C .

Its second dimension must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a . Constraints:
 $lda \geq \max(1, r)$ if $\text{vect} = \text{'Q'}$;
 $lda \geq \max(1, \min(r, k))$ if $\text{vect} = \text{'P'}$.

lrc INTEGER. The leading dimension of *c*; *lrc* $\geq \max(1, m)$.

tau COMPLEX for *cunmbr*
DOUBLE COMPLEX for *zunmbr*.
Array, DIMENSION at least $\max(1, \min(r, k))$.
For *vect* = 'Q', the array *tauq* as returned by ?gebrd. For *vect* = 'P', the array *taup* as returned by ?gebrd.

lwork INTEGER. The size of the *work* array.
lwork $\geq \max(1, n)$ if *side* = 'L';
lwork $\geq \max(1, m)$ if *side* = 'R'.
lwork ≥ 1 if *n*=0 or *m*=0.
For optimum performance *lwork* $\geq \max(1, n*nb)$ if *side* = 'L', and *lwork* $\geq \max(1, m*nb)$ if *side* = 'R', where *nb* is the optimal blocksize. (*nb* = 0 if *m* = 0 or *n* = 0.)
If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.
See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c Overwritten by the product Q^*C , $Q^{H*}C$, C^*Q , C^*Q^H , P^*C , $P^{H*}C$, C^*P , or C^*P^H , as specified by *vect*, *side*, and *trans*.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *unmbr* interface are the following:

a Holds the matrix *A* of size $(r, \min(nq, k))$ where
r = *nq*, if *vect* = 'Q',
r = $\min(nq, k)$, if *vect* = 'P',
nq = *m*, if *side* = 'L',
nq = *n*, if *side* = 'R',
k = *m*, if *vect* = 'P',

	$k = n$, if <code>vect = 'Q'</code> .
<code>tau</code>	Holds the vector of length $\min(nq, k)$.
<code>c</code>	Holds the matrix C of size (m, n) .
<code>vect</code>	Must be ' <code>Q</code> ' or ' <code>P</code> '. The default value is ' <code>Q</code> '.
<code>side</code>	Must be ' <code>L</code> ' or ' <code>R</code> '. The default value is ' <code>L</code> '.
<code>trans</code>	Must be ' <code>N</code> ' or ' <code>C</code> '. The default value is ' <code>N</code> '.

Application Notes

For better performance, use

`lwork = n*blocksize` for `side = 'L'`, or

`lwork = m*blocksize` for `side = 'R'`,

where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of `lwork` for the first run, or set `lwork = -1`.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If `lwork = -1`, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if `lwork` is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\epsilon) * \|C\|_2$.

The total number of floating-point operations is approximately

$8*n*k(2*m - k)$ if `side = 'L'` and $m \geq k$;

$8*m*k(2*n - k)$ if `side = 'R'` and $n \geq k$;

$8*m^2*n$ if `side = 'L'` and $m < k$;

$8*n^2*m$ if `side = 'R'` and $n < k$.

The real counterpart of this routine is [ormbr](#).

?bdsqr

Computes the singular value decomposition of a general matrix that has been reduced to bidiagonal form.

Syntax

FORTRAN 77:

```
call sbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info)
call dbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info)
call cbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, rwork, info)
call zbdsqr(uplo, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, rwork, info)
```

Fortran 95:

```
call rbdsqr(d, e [,vt] [,u] [,c] [,uplo] [,info])
call bdsqr(d, e [,vt] [,u] [,c] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_sbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncvt,
lapack_int nru, lapack_int ncc, float* d, float* e, float* vt, lapack_int ldvt, float*
u, lapack_int ldu, float* c, lapack_int ldc );
lapack_int LAPACKE_dbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncvt,
lapack_int nru, lapack_int ncc, double* d, double* e, double* vt, lapack_int ldvt,
double* u, lapack_int ldu, double* c, lapack_int ldc );
lapack_int LAPACKE_cbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncvt,
lapack_int nru, lapack_int ncc, float* d, float* e, lapack_complex_float* vt,
lapack_int ldvt, lapack_complex_float* u, lapack_int ldu, lapack_complex_float* c,
lapack_int ldc );
lapack_int LAPACKE_zbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncvt,
lapack_int nru, lapack_int ncc, double* d, double* e, lapack_complex_double* vt,
lapack_int ldvt, lapack_complex_double* u, lapack_int ldu, lapack_complex_double* c,
lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the singular values and, optionally, the right and/or left singular vectors from the [Singular Value Decomposition](#) (SVD) of a real n -by- n (upper or lower) bidiagonal matrix B using the implicit zero-shift QR algorithm. The SVD of B has the form $B = Q * S * P^H$ where S is the diagonal matrix of singular values, Q is an orthogonal matrix of left singular vectors, and P is an orthogonal matrix of right singular vectors. If left singular vectors are requested, this subroutine actually returns $U * Q$ instead of Q , and, if right singular vectors are requested, this subroutine returns $P_H * VT$ instead of P_H , for given real/complex input matrices U and VT . When U and VT are the orthogonal/unitary matrices that reduce a general matrix A to bidiagonal form: $A = U * B * VT$, as computed by `?gebrd`, then

$$A = (U * Q) * S * (P^H * VT)$$

is the SVD of A . Optionally, the subroutine may also compute $Q_H * C$ for a given real/complex input matrix C .

See also [lasq1](#), [lasq2](#), [lasq3](#), [lasq4](#), [lasq5](#), [lasq6](#) used by this routine.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	If <i>uplo</i> = 'U', B is an upper bidiagonal matrix.
	If <i>uplo</i> = 'L', B is a lower bidiagonal matrix.

n INTEGER. The order of the matrix B ($n \geq 0$).

<i>ncvt</i>	INTEGER. The number of columns of the matrix VT , that is, the number of right singular vectors ($ncvt \geq 0$). Set $ncvt = 0$ if no right singular vectors are required.
<i>nru</i>	INTEGER. The number of rows in U , that is, the number of left singular vectors ($nru \geq 0$). Set $nru = 0$ if no left singular vectors are required.
<i>ncc</i>	INTEGER. The number of columns in the matrix C used for computing the product $Q^H * C$ ($ncc \geq 0$). Set $ncc = 0$ if no matrix C is supplied.
<i>d, e</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i> (*) contains the diagonal elements of B . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the $(n-1)$ off-diagonal elements of B . The dimension of <i>e</i> must be at least $\max(1, n)$.
<i>work</i>	REAL for sbdsqr DOUBLE PRECISION for dbdsqr. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 4*n)$.
<i>rwork</i>	REAL for cbdsqr DOUBLE PRECISION for zbdsqr. <i>rwork</i> (*) is a workspace array. If $ncvt = nru = ncc = 0$, the dimension of <i>rwork</i> must be $\max(1, 2*n)$. Otherwise, the dimension of <i>rwork</i> must be $\max(1, 4*n-4)$.
<i>vt, u, c</i>	REAL for sbdsqr DOUBLE PRECISION for dbdsqr COMPLEX for cbdsqr DOUBLE COMPLEX for zbdsqr. Arrays: <i>vt</i> (<i>ldvt</i> ,*) contains an n -by- $ncvt$ matrix VT . The second dimension of <i>vt</i> must be at least $\max(1, ncvt)$. <i>vt</i> is not referenced if $ncvt = 0$. <i>u</i> (<i>ldu</i> ,*) contains an nru by n unit matrix U . The second dimension of <i>u</i> must be at least $\max(1, n)$. <i>u</i> is not referenced if $nru = 0$. <i>c</i> (<i>ldc</i> ,*) contains the matrix C for computing the product $Q^H * C$.

The second dimension of c must be at least $\max(1, ncc)$. The array is not referenced if $ncc = 0$.

$ldvt$ INTEGER. The leading dimension of vt . Constraints:

$ldvt \geq \max(1, n)$ if $ncvt > 0$;

$ldvt \geq 1$ if $ncvt = 0$.

ldu INTEGER. The leading dimension of u . Constraint:

$ldu \geq \max(1, nru)$.

ldc INTEGER. The leading dimension of c . Constraints:

$ldc \geq \max(1, n)$ if $ncc > 0$; $ldc \geq 1$ otherwise.

Output Parameters

d	On exit, if $info = 0$, overwritten by the singular values in decreasing order (see $info$).
e	On exit, if $info = 0$, e is destroyed. See also $info$ below.
c	Overwritten by the product $Q^H * C$.
vt	On exit, this array is overwritten by $P^H * VT$.
u	On exit, this array is overwritten by $U * Q$.
$info$	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th parameter had an illegal value.</p> <p>If $info > i$,</p> <p>If $ncvt = nru = ncc = 0$,</p> <ul style="list-style-type: none"> • $info = 1$, a split was marked by a positive value in e • $info = 2$, the current block of z not diagonalized after $30*n$ iterations (in the inner while loop) • $info = 3$, termination criterion of the outer while loop is not met (the program created more than n unreduced blocks). <p>In all other cases when $ncvt = nru = ncc = 0$, the algorithm did not converge; d and e contain the elements of a bidiagonal matrix that is orthogonally similar to the input matrix B; if $info = i$, i elements of e have not converged to zero.</p>

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `bdsqr` interface are the following:

d Holds the vector of length (n).

<i>e</i>	Holds the vector of length (<i>n</i>).
<i>vt</i>	Holds the matrix VT of size (<i>n</i> , <i>ncvt</i>).
<i>u</i>	Holds the matrix U of size (<i>nru</i> , <i>n</i>).
<i>c</i>	Holds the matrix C of size (<i>n</i> , <i>ncc</i>).
<i>uplo</i>	Must be ' <i>U</i> ' or ' <i>L</i> '. The default value is ' <i>U</i> '.
<i>ncvt</i>	If argument <i>vt</i> is present, then <i>ncvt</i> is equal to the number of columns in matrix VT ; otherwise, <i>ncvt</i> is set to zero.
<i>nru</i>	If argument <i>u</i> is present, then <i>nru</i> is equal to the number of rows in matrix U ; otherwise, <i>nru</i> is set to zero.
<i>ncc</i>	If argument <i>c</i> is present, then <i>ncc</i> is equal to the number of columns in matrix C ; otherwise, <i>ncc</i> is set to zero.

Note that two variants of Fortran 95 interface for `bdsqr` routine are needed because of an ambiguous choice between real and complex cases appear when *vt*, *u*, and *c* are omitted. Thus, the name `rbdsqr` is used in real cases (single or double precision), and the name `bdsqr` is used in complex cases (single or double precision).

Application Notes

Each singular value and singular vector is computed to high relative accuracy. However, the reduction to bidiagonal form (prior to calling the routine) may decrease the relative accuracy in the small singular values of the original matrix if its singular values vary widely in magnitude.

If s_i is an exact singular value of B , and s_i is the corresponding computed value, then

$$|s_i - \sigma_i| \leq p(m, n) * \varepsilon * \sigma_i$$

where $p(m, n)$ is a modestly increasing function of *m* and *n*, and ε is the machine precision.

If only singular values are computed, they are computed more accurately than when some singular vectors are also computed (that is, the function $p(m, n)$ is smaller).

If u_i is the corresponding exact left singular vector of B , and w_i is the corresponding computed left singular vector, then the angle $\theta(u_i, w_i)$ between them is bounded as follows:

$$\theta(u_i, w_i) \leq p(m, n) * \varepsilon / \min_{i \neq j} (|\sigma_i - \sigma_j| / |\sigma_i + \sigma_j|).$$

Here $\min_{i \neq j} (|\sigma_i - \sigma_j| / |\sigma_i + \sigma_j|)$ is the *relative gap* between σ_i and the other singular values. A similar error bound holds for the right singular vectors.

The total number of real floating-point operations is roughly proportional to n^2 if only the singular values are computed. About $6n^2 * nru$ additional operations ($12n^2 * nru$ for complex flavors) are required to compute the left singular vectors and about $6n^2 * ncvt$ operations ($12n^2 * ncvt$ for complex flavors) to compute the right singular vectors.

?bdsdc

Computes the singular value decomposition of a real bidiagonal matrix using a divide and conquer method.

Syntax

FORTRAN 77:

```
call sbdsdc(uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)
call dbdsdc(uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq, work, iwork, info)
```

Fortran 95:

```
call bdsdc(d, e [,u] [,vt] [,q] [,iq] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>bdsdc( int matrix_layout, char uplo, char compq, lapack_int n,
<datatype>* d, <datatype>* e, <datatype>* u, lapack_int ldu, <datatype>* vt,
lapack_int ldvt, <datatype>* q, lapack_int* iq );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the [Singular Value Decomposition](#) (SVD) of a real n -by- n (upper or lower) bidiagonal matrix B : $B = U\Sigma V^T$, using a divide and conquer method, where Σ is a diagonal matrix with non-negative diagonal elements (the singular values of B), and U and V are orthogonal matrices of left and right singular vectors, respectively. ?bdsdc can be used to compute all singular values, and optionally, singular vectors or singular vectors in compact form.

This routine

uses ?lasd0, ?lasd1, ?lasd2, ?lasd3, ?lasd4, ?lasd5, ?lasd6, ?lasd7, ?lasd8, ?lasd9, ?lasda, ?lasdq, ?lasdt.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	If <i>uplo</i> = 'U', B is an upper bidiagonal matrix.
	If <i>uplo</i> = 'L', B is a lower bidiagonal matrix.
<i>compq</i>	CHARACTER*1. Must be 'N', 'P', or 'I'.
	If <i>compq</i> = 'N', compute singular values only.
	If <i>compq</i> = 'P', compute singular values and compute singular vectors in compact form.
	If <i>compq</i> = 'I', compute singular values and singular vectors.
<i>n</i>	INTEGER. The order of the matrix B ($n \geq 0$).
<i>d, e, work</i>	REAL for sbdsdc DOUBLE PRECISION for dbdsdc. Arrays: <i>d</i> (*) contains the n diagonal elements of the bidiagonal matrix B . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the off-diagonal elements of the bidiagonal matrix B . The dimension of <i>e</i> must be at least $\max(1, n)$.

work(*) is a workspace array.

The dimension of *work* must be at least:

$\max(1, 4*n)$, if *compq* = 'N';

$\max(1, 6*n)$, if *compq* = 'P';

$\max(1, 3*n^2+4*n)$, if *compq* = 'I'.

ldu

INTEGER. The leading dimension of the output array *u*; *ldu* ≥ 1 .

If singular vectors are desired, then *ldu* $\geq \max(1, n)$.

ldvt

INTEGER. The leading dimension of the output array *vt*; *ldvt* ≥ 1 .

If singular vectors are desired, then *ldvt* $\geq \max(1, n)$.

iwork

INTEGER. Workspace array, dimension at least $\max(1, 8*n)$.

Output Parameters

d

If *info* = 0, overwritten by the singular values of *B*.

e

On exit, *e* is overwritten.

u, *vt*, *q*

REAL for sbdsdc

DOUBLE PRECISION for dbdsdc.

Arrays: *u*(*ldu*, *), *vt*(*ldvt*, *), *q*(*).

If *compq* = 'I', then on exit *u* contains the left singular vectors of the bidiagonal matrix *B*, unless *info* $\neq 0$ (seeinfo). For other values of *compq*, *u* is not referenced.

The second dimension of *u* must be at least $\max(1,n)$.

If *compq* = 'I', then on exit *vt*^T contains the right singular vectors of the bidiagonal matrix *B*, unless *info* $\neq 0$ (seeinfo). For other values of *compq*, *vt* is not referenced. The second dimension of *vt* must be at least $\max(1,n)$.

If *compq* = 'P', then on exit, if *info* = 0, *q* and *iq* contain the left and right singular vectors in a compact form. Specifically, *q* contains all the REAL (for sbdsdc) or DOUBLE PRECISION (for dbdsdc) data for singular vectors. For other values of *compq*, *q* is not referenced. See *Application notes* for details.

iq

INTEGER.

Array: *iq*(*).

If *compq* = 'P', then on exit, if *info* = 0, *q* and *iq* contain the left and right singular vectors in a compact form. Specifically, *iq* contains all the INTEGER data for singular vectors. For other values of *compq*, *iq* is not referenced. See *Application notes* for details.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If $\text{info} = i$, the algorithm failed to compute a singular value. The update process of divide and conquer failed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `bdsdc` interface are the following:

d	Holds the vector of length n .
e	Holds the vector of length n .
u	Holds the matrix U of size (n,n) .
vt	Holds the matrix VT of size (n,n) .
q	Holds the vector of length (ldq) , where $ldq \geq n * (11 + 2 * smlsiz + 8 * \text{int}(\log_2(n / (smlsiz + 1))))$ and $smlsiz$ is returned by <code>ilaenv</code> and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25).
$compq$	Restored based on the presence of arguments u , vt , q , and iq as follows: $compq = 'N'$, if none of u , vt , q , and iq are present, $compq = 'I'$, if both u and vt are present. Arguments u and vt must either be both present or both omitted, $compq = 'P'$, if both q and iq are present. Arguments q and iq must either be both present or both omitted. Note that there will be an error condition if all of u , vt , q , and iq arguments are present simultaneously.

See Also

[?lasd0](#)
[?lasd1](#)
[?lasd2](#)
[?lasd3](#)
[?lasd4](#)
[?lasd5](#)
[?lasd6](#)
[?lasd7](#)
[?lasd8](#)
[?lasd9](#)
[?lasda](#)
[?lasdq](#)
[?lasdt](#)

Symmetric Eigenvalue Problems

Symmetric eigenvalue problems are posed as follows: given an n -by- n real symmetric or complex Hermitian matrix A , find the *eigenvalues* λ and the corresponding *eigenvectors* z that satisfy the equation $Az = \lambda z$ (or, equivalently, $z^H A = \lambda z^H$).

In such eigenvalue problems, all n eigenvalues are real not only for real symmetric but also for complex Hermitian matrices A , and there exists an orthonormal system of n eigenvectors. If A is a symmetric or Hermitian positive-definite matrix, all eigenvalues are positive.

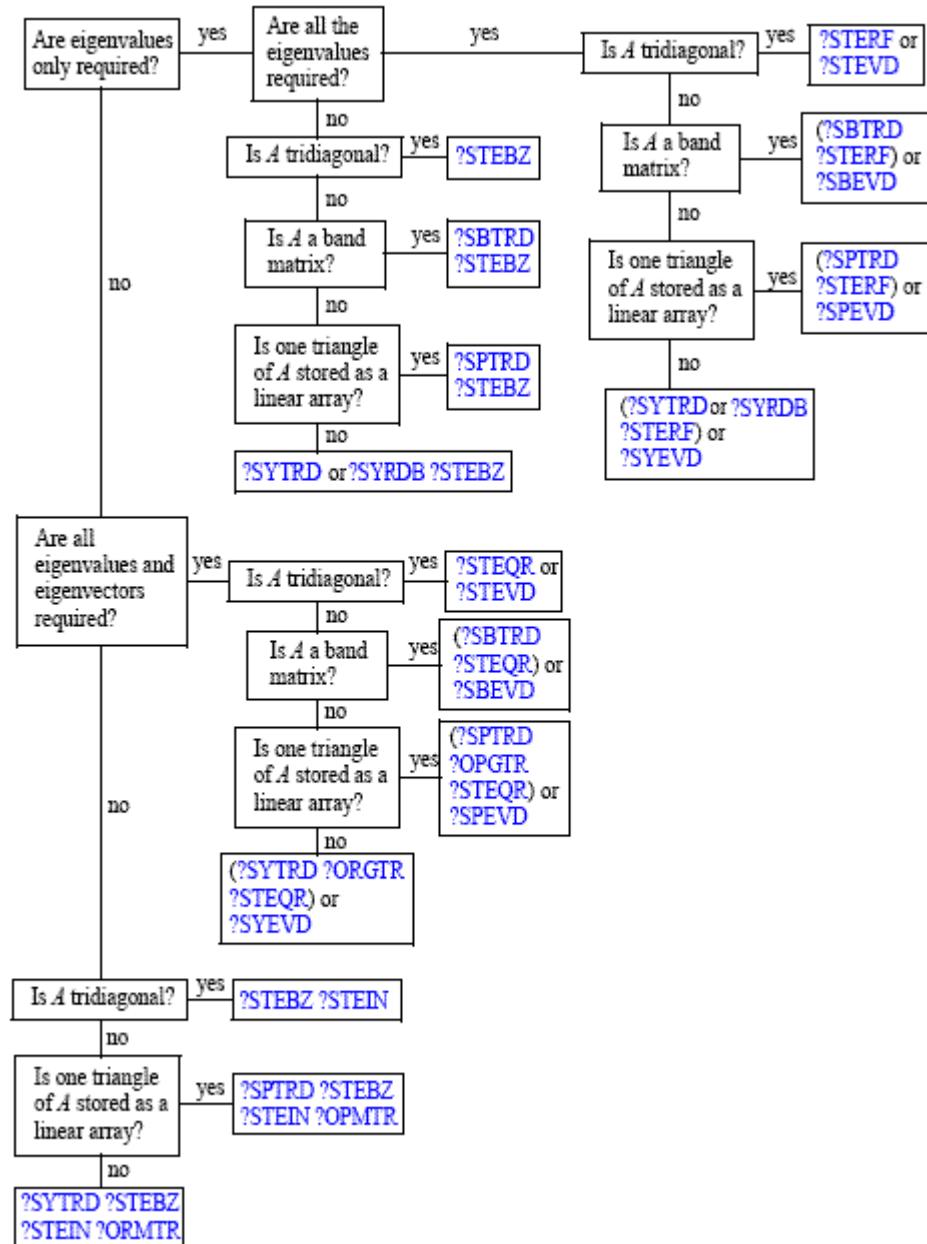
To solve a symmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to tridiagonal form and then solve the eigenvalue problem with the tridiagonal matrix obtained. LAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines (for FORTRAN 77 interface) are listed in [Table "Computational Routines for Solving Symmetric Eigenvalue Problems"](#). The corresponding routine names in the Fortran 95 interface are without the first symbol.

There are different routines for symmetric eigenvalue problems, depending on whether you need all eigenvectors or only some of them or eigenvalues only, whether the matrix A is positive-definite or not, and so on.

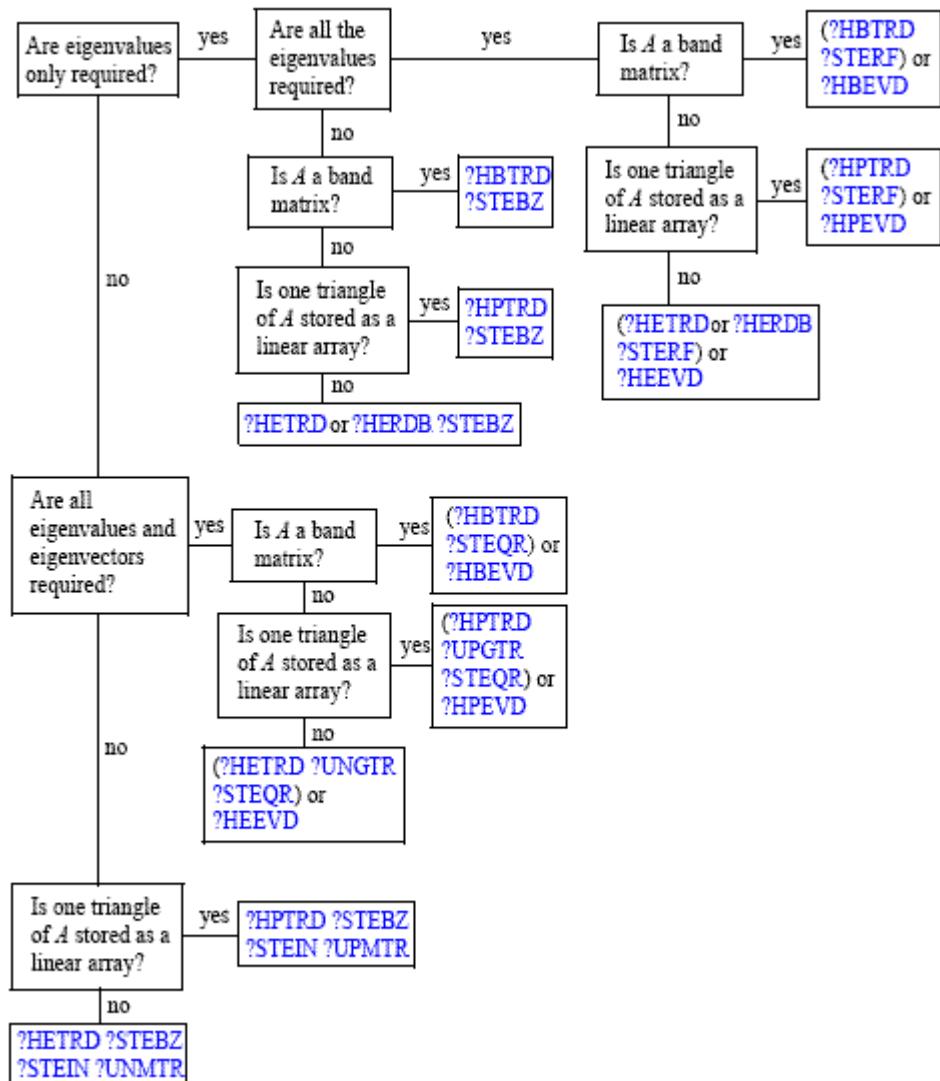
These routines are based on three primary algorithms for computing eigenvalues and eigenvectors of symmetric problems: the divide and conquer algorithm, the QR algorithm, and bisection followed by inverse iteration. The divide and conquer algorithm is generally more efficient and is recommended for computing all eigenvalues and eigenvectors. Furthermore, to solve an eigenvalue problem using the divide and conquer algorithm, you need to call only one routine. In general, more than one routine has to be called if the QR algorithm or bisection followed by inverse iteration is used.

The decision tree in [Figure "Decision Tree: Real Symmetric Eigenvalue Problems"](#) will help you choose the right routine or sequence of routines for eigenvalue problems with real symmetric matrices. [Figure "Decision Tree: Complex Hermitian Eigenvalue Problems"](#) presents a similar decision tree for complex Hermitian matrices.

Decision Tree: Real Symmetric Eigenvalue Problems



Decision Tree: Complex Hermitian Eigenvalue Problems



Computational Routines for Solving Symmetric Eigenvalue Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to tridiagonal form $A = QTQ^H$ (full storage)	sytrd syrdb	hetrd herdb
Reduce to tridiagonal form $A = QTQ^H$ (packed storage)	sptrd	hptrd
Reduce to tridiagonal form $A = QTQ^H$ (band storage).	sbtrd	hbtrd
Generate matrix Q (full storage)	orgtr	ungtr
Generate matrix Q (packed storage)	opgtr	upgtr
Apply matrix Q (full storage)	ormtr	unmtr
Apply matrix Q (packed storage)	opmtr	upmtr

Operation	Real symmetric matrices	Complex Hermitian matrices
Find all eigenvalues of a tridiagonal matrix T	<code>sterf</code>	
Find all eigenvalues and eigenvectors of a tridiagonal matrix T	<code>steqr stedc</code>	<code>steqr stedc</code>
Find all eigenvalues and eigenvectors of a tridiagonal positive-definite matrix T .	<code>pteqr</code>	<code>pteqr</code>
Find selected eigenvalues of a tridiagonal matrix T	<code>stebz steqr</code>	<code>stegr</code>
Find selected eigenvectors of a tridiagonal matrix T	<code>stein stegr</code>	<code>stein stegr</code>
Find selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix T	<code>stemr</code>	<code>stemr</code>
Compute the reciprocal condition numbers for the eigenvectors	<code>disna</code>	<code>disna</code>

?sytrd*Reduces a real symmetric matrix to tridiagonal form.***Syntax****FORTRAN 77:**

```
call ssytrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call dsytrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
```

Fortran 95:

```
call sytrd(a, tau [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>sytrd( int matrix_layout, char uplo, lapack_int n, <datatype>* a,
lapack_int lda, <datatype>* d, <datatype>* e, <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q^T Q^T$. The orthogonal matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation (see *Application Notes* below).

This routine calls `latrd` to reduce a real symmetric matrix to tridiagonal form by an orthogonal similarity transformation.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> .
	If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> (<i>n</i> ≥ 0).
<i>a, work</i>	REAL for ssytrd DOUBLE PRECISION for dsytrd. <i>a</i> (<i>lda</i> ,*) is an array containing either upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> , and the strictly lower triangular part of <i>A</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i> , and the strictly upper triangular part of <i>A</i> is not referenced. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array (<i>lwork</i> ≥ <i>n</i>). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	On exit, if <i>uplo</i> = 'U', the diagonal and first superdiagonal of <i>A</i> are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i> , and the elements above the first superdiagonal, with the array <i>tau</i> , represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors; if <i>uplo</i> = 'L', the diagonal and first subdiagonal of <i>A</i> are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i> , and the elements below the first subdiagonal, with the array <i>tau</i> , represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors.
<i>d, e, tau</i>	REAL for ssytrd DOUBLE PRECISION for dsytrd. Arrays: <i>d</i> (*) contains the diagonal elements of the matrix <i>T</i> . The dimension of <i>d</i> must be at least $\max(1, n)$.

$e(*)$ contains the off-diagonal elements of T .

The dimension of e must be at least $\max(1, n-1)$.

$\tau_{\text{au}}(*)$ stores further details of the orthogonal matrix Q in the first $n-1$ elements. $\tau_{\text{au}}(n)$ is used as workspace.

The dimension of τ_{au} must be at least $\max(1, n)$.

$\text{work}(1)$
If $\text{info}=0$, on exit $\text{work}(1)$ contains the minimum value of lwork required for optimum performance. Use this lwork for subsequent runs.

info INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sytrd` interface are the following:

a Holds the matrix A of size (n,n) .

τ_{au} Holds the vector of length $(n-1)$.

uplo Must be ' U ' or ' L '. The default value is ' U '.

Note that diagonal (d) and off-diagonal (e) elements of the matrix T are omitted because they are kept in the matrix A on exit.

Application Notes

For better performance, try using $\text{lwork} = n * \text{blocksize}$, where blocksize is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of lwork for the first run, or set $\text{lwork} = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array work on exit. Use this value ($\text{work}(1)$) for subsequent runs.

If $\text{lwork} = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (work). This operation is called a workspace query.

Note that if lwork is less than the minimal required value and is not equal to -1 , then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix T is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n) * \varepsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

`orgtr` to form the computed matrix Q explicitly

`ormtr` to multiply a real matrix by Q .

The complex counterpart of this routine is [hetrd](#).

?syrdb

Reduces a real symmetric matrix to tridiagonal form with Successive Bandwidth Reduction approach.

Syntax

FORTRAN 77:

```
call ssyrd(b, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
call dsyrd(b, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine reduces a real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q^*T^*Q$ and optionally multiplies matrix Z by Q , or simply forms Q .

This routine reduces a full symmetric matrix A to the banded symmetric matrix B , and then to the tridiagonal symmetric matrix T with a Successive Bandwidth Reduction approach after C. Bischof's works (see for instance, [[Bischof00](#)]). ?syrd is functionally close to ?sytrd routine but the tridiagonal form may differ from those obtained by ?sytrd. Unlike ?sytrd, the orthogonal matrix Q cannot be restored from the details of matrix A on exit.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only A is reduced to T . If <i>jobz</i> = 'V', then A is reduced to T and A contains Q on exit. If <i>jobz</i> = 'U', then A is reduced to T and Z contains Z^*Q on exit.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', a stores the upper triangular part of A . If <i>uplo</i> = 'L', a stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>kd</i>	INTEGER. The bandwidth of the banded matrix B ($kd \geq 1$, $kd \leq n-1$).
<i>a, z, work</i>	REAL for ssyrd. DOUBLE PRECISION for dsyrd. $a(1:da, :)$ is an array containing either upper or lower triangular part of the matrix A , as specified by <i>uplo</i> . The second dimension of a must be at least $\max(1, n)$. $z(1:dz, :)$, the second dimension of z must be at least $\max(1, n)$. If <i>jobz</i> = 'U', then the matrix z is multiplied by Q .

If $\text{jobz} = \text{'N'}$ or 'V' , then z is not referenced.

$\text{work}(lwork)$ is a workspace array.

lda

INTEGER. The leading dimension of a ; at least $\max(1, n)$.

ldz

INTEGER. The leading dimension of z ; at least $\max(1, n)$. Not referenced if $\text{jobz} = \text{'N'}$

lwork

INTEGER. The size of the work array ($\text{lwork} \geq (2kd+1)n+kd$).

If $\text{lwork} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the work array, returns this value as the first entry of the work array, and no error message related to lwork is issued by xerbla .

See *Application Notes* for the suggested value of lwork .

Output Parameters

a

If $\text{jobz} = \text{'V'}$, then overwritten by Q matrix.

If $\text{jobz} = \text{'N'}$ or 'U' , then overwritten by the banded matrix B and details of the orthogonal matrix Q_B to reduce A to B as specified by uplo .

z

On exit,

if $\text{jobz} = \text{'U'}$, then the matrix z is overwritten by Z^*Q .

If $\text{jobz} = \text{'N'}$ or 'V' , then z is not referenced.

d, e, tau

DOUBLE PRECISION.

Arrays:

$d(*)$ contains the diagonal elements of the matrix T .

The dimension of d must be at least $\max(1, n)$.

$e(*)$ contains the off-diagonal elements of T .

The dimension of e must be at least $\max(1, n-1)$.

$tau(*)$ stores further details of the orthogonal matrix Q .

The dimension of tau must be at least $\max(1, n-kd-1)$.

$\text{work}(1)$

If $\text{info}=0$, on exit $\text{work}(1)$ contains the minimum value of lwork required for optimum performance. Use this lwork for subsequent runs.

info

INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th parameter had an illegal value.

Application Notes

For better performance, try using $\text{lwork} = n * (3 * kd + 3)$.

If it is not clear how much workspace to supply, use a generous value of lwork for the first run, or set $\text{lwork} = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

For better performance, try using *kd* equal to 40 if $n \leq 2000$ and 64 otherwise.

Try using ?syrd instead of ?sytrd on large matrices obtaining only eigenvalues - when no eigenvectors are needed, especially in multi-threaded environment. ?syrd becomes faster beginning approximately with $n = 1000$, and much faster at larger matrices with a better scalability than ?sytrd.

Avoid applying ?syrd for computing eigenvectors due to the two-step reduction, that is, the number of operations needed to apply orthogonal transformations to *Z* is doubled compared to the traditional one-step reduction. In that case it is better to apply ?sytrd and ?ormtr/?orgtr to obtain tridiagonal form along with the orthogonal transformation matrix *Q*.

?herdb

Reduces a complex Hermitian matrix to tridiagonal form with Successive Bandwidth Reduction approach.

Syntax

FORTRAN 77:

```
call cherdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
call zherdb(jobz, uplo, n, kd, a, lda, d, e, tau, z, ldz, work, lwork, info)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine reduces a complex Hermitian matrix *A* to symmetric tridiagonal form *T* by a unitary similarity transformation: $A = Q^*T^*Q$ and optionally multiplies matrix *Z* by *Q*, or simply forms *Q*.

This routine reduces a full symmetric matrix *A* to the banded symmetric matrix *B*, and then to the tridiagonal symmetric matrix *T* with a Successive Bandwidth Reduction approach after C. Bischof's works (see for instance, [Bischof00]). ?herdb is functionally close to ?hetrd routine but the tridiagonal form may differ from those obtained by ?hetrd. Unlike ?hetrd, the orthogonal matrix *Q* cannot be restored from the details of matrix *A* on exit.

Input Parameters

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only <i>A</i> is reduced to <i>T</i> . If <i>jobz</i> = 'V', then <i>A</i> is reduced to <i>T</i> and <i>A</i> contains <i>Q</i> on exit. If <i>jobz</i> = 'U', then <i>A</i> is reduced to <i>T</i> and <i>Z</i> contains <i>Z*Q</i> on exit.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> .

If $uplo = 'L'$, a stores the lower triangular part of A .

n INTEGER. The order of the matrix A ($n \geq 0$).

kd INTEGER. The bandwidth of the banded matrix B ($kd \geq 1$, $kd \leq n-1$).

$a, z, work$ COMPLEX for cherdb.

DOUBLE COMPLEX for zherdb.

$a(Ida,*)$ is an array containing either upper or lower triangular part of the matrix A , as specified by $uplo$.

The second dimension of a must be at least $\max(1, n)$.

$z(Idz,*)$, the second dimension of z must be at least $\max(1, n)$.

If $jobz = 'U'$, then the matrix z is multiplied by Q .

If $jobz = 'N'$ or ' V ', then z is not referenced.

$work(lwork)$ is a workspace array.

lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.

ldz INTEGER. The leading dimension of z ; at least $\max(1, n)$. Not referenced if $jobz = 'N'$

$lwork$ INTEGER. The size of the $work$ array ($lwork \geq (2kd+1)n+kd$).

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by **xerbla**.

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a If $jobz = 'V'$, then overwritten by Q matrix.

If $jobz = 'N'$ or ' U ', then overwritten by the banded matrix B and details of the unitary matrix Q_B to reduce A to B as specified by $uplo$.

z On exit,

if $jobz = 'U'$, then the matrix z is overwritten by Z^*Q .

If $jobz = 'N'$ or ' V ', then z is not referenced.

d, e REAL for cherdb.

DOUBLE PRECISION for zherdb.

Arrays:

$d(*)$ contains the diagonal elements of the matrix T .

The dimension of d must be at least $\max(1, n)$.

$e(*)$ contains the off-diagonal elements of T .

The dimension of e must be at least $\max(1, n-1)$.

$tau(*)$ stores further details of the orthogonal matrix Q .

The dimension of *tau* must be at least $\max(1, n-kd-1)$.

<i>tau</i>	COMPLEX for <i>cherdb</i> . DOUBLE COMPLEX for <i>zherdb</i> .
	Array, DIMENSION at least $\max(1, n-1)$
	Stores further details of the unitary matrix Q_B . The dimension of <i>tau</i> must be at least $\max(1, n-kd-1)$.
<i>work(1)</i>	If <i>info</i> =0, on exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Application Notes

For better performance, try using $lwork = n * (3 * kd + 3)$.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

For better performance, try using *kd* equal to 40 if $n \leq 2000$ and 64 otherwise.

Try using *?herdb* instead of *?hetrd* on large matrices obtaining only eigenvalues - when no eigenvectors are needed, especially in multi-threaded environment. *?herdb* becomes faster beginning approximately with $n = 1000$, and much faster at larger matrices with a better scalability than *?hetrd*.

Avoid applying *?herdb* for computing eigenvectors due to the two-step reduction, that is, the number of operations needed to apply orthogonal transformations to *Z* is doubled compared to the traditional one-step reduction. In that case it is better to apply *?hetrd* and *?unmtr*/*?ungtr* to obtain tridiagonal form along with the unitary transformation matrix *Q*.

?orgtr

Generates the real orthogonal matrix *Q* determined by *?sytrd*.

Syntax

FORTRAN 77:

```
call sorgtr(uplo, n, a, lda, tau, work, lwork, info)
call dorgtr(uplo, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orgtr(a, tau [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>orgtr( int matrix_layout, char uplo, lapack_int n, <datatype>* a,
lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine explicitly generates the n -by- n orthogonal matrix Q formed by [sytrd](#) when reducing a real symmetric matrix A to tridiagonal form: $A = Q^T T Q$. Use this routine after a call to [?sytrd](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Use the same <i>uplo</i> as supplied to ?sytrd .
<i>n</i>	INTEGER. The order of the matrix Q ($n \geq 0$).
<i>a</i> , <i>tau</i> , <i>work</i>	REAL for sorgtr DOUBLE PRECISION for dorgtr. Arrays: <i>a</i> (<i>lda</i> ,*) is the array <i>a</i> as returned by ?sytrd . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>tau</i> (*) is the array <i>tau</i> as returned by ?sytrd . The dimension of <i>tau</i> must be at least $\max(1, n-1)$. <i>work</i> is a workspace array, its dimension $\max(1, \textit{lwork})$. <i>lda</i> INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$. <i>lwork</i> INTEGER. The size of the <i>work</i> array ($\textit{lwork} \geq n$). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the orthogonal matrix Q .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `orgtr` interface are the following:

<code>a</code>	Holds the matrix A of size (n,n) .
<code>tau</code>	Holds the vector of length $(n-1)$.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

For better performance, try using `lwork = (n-1)*blocksize`, where `blocksize` is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of `lwork` for the first run or set `lwork = -1`.

If you choose the first option and set any of admissible `lwork` sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is [ungtr](#).

?ormtr

Multiples a real matrix by the real orthogonal matrix Q determined by ?sytrd.

Syntax

FORTRAN 77:

```
call sormtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call dormtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>ormtr( int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const <datatype>* a, lapack_int lda, const <datatype>*
tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine multiplies a real matrix C by Q or Q^T , where Q is the orthogonal matrix Q formed by [sytrd](#) when reducing a real symmetric matrix A to tridiagonal form: $A = Q^T T Q$. Use this routine after a call to [?sytrd](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q^* C$, $Q^T C$, $C Q$, or $C^* Q^T$ (overwriting the result on C).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

In the descriptions below, r denotes the order of Q :

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If <i>side</i> = 'L', Q or Q^T is applied to C from the left. If <i>side</i> = 'R', Q or Q^T is applied to C from the right.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?sytrd .
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If <i>trans</i> = 'N', the routine multiplies C by Q . If <i>trans</i> = 'T', the routine multiplies C by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>a</i> , <i>c</i> , <i>tau</i> , <i>work</i>	REAL for sormtr DOUBLE PRECISION for dormtr <i>a</i> (<i>lda</i> ,*) and <i>tau</i> are the arrays returned by ?sytrd . The second dimension of <i>a</i> must be at least $\max(1, r)$. The dimension of <i>tau</i> must be at least $\max(1, r-1)$. <i>c</i> (<i>ldc</i> ,*) contains the matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work</i> is a workspace array, its dimension $\max(1, \text{ lwork})$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $\text{lda} \geq \max(1, r)$.
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> ; $\text{ldc} \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints:

```
lwork ≥ max(1, n) if side = 'L';
lwork ≥ max(1, m) if side = 'R'.
```

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , Q^T*C , C^*Q , or C^*Q^T (as specified by <i>side</i> and <i>trans</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormtr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>r,r</i>). <i>r</i> = <i>m</i> if <i>side</i> = 'L'. <i>r</i> = <i>n</i> if <i>side</i> = 'R'.
<i>tau</i>	Holds the vector of length (<i>r-1</i>).
<i>c</i>	Holds the matrix <i>C</i> of size (<i>m,n</i>).
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

For better performance, try using *lwork* = *n***blocksize* for *side* = 'L', or *lwork* = *m***blocksize* for *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix *E* such that $\|E\|^2 = O(\epsilon) * \|C\|^2$.

The total number of floating-point operations is approximately $2*m^2*n$, if *side* = 'L', or $2*n^2*m$, if *side* = 'R'.

The complex counterpart of this routine is [unmtr](#).

?hetrd

Reduces a complex Hermitian matrix to tridiagonal form.

Syntax

FORTRAN 77:

```
call chetrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
call zhetrd(uplo, n, a, lda, d, e, tau, work, lwork, info)
```

Fortran 95:

```
call hetrd(a, tau [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_chetrd( int matrix_layout, char uplo, lapack_int n,
lapack_complex_float* a, lapack_int lda, float* d, float* e, lapack_complex_float*
tau );
lapack_int LAPACKE_zhetrd( int matrix_layout, char uplo, lapack_int n,
lapack_complex_double* a, lapack_int lda, double* d, double* e, lapack_complex_double*
tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces a complex Hermitian matrix *A* to symmetric tridiagonal form *T* by a unitary similarity transformation: $A = Q*T*Q^H$. The unitary matrix *Q* is not formed explicitly but is represented as a product of *n*-1 elementary reflectors. Routines are provided to work with *Q* in this representation. (They are described later in this section.)

This routine calls [latrd](#) to reduce a complex Hermitian matrix *A* to Hermitian tridiagonal form by a unitary similarity transformation.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> (<i>n</i> \geq 0).
<i>a, work</i>	COMPLEX for chetrd DOUBLE COMPLEX for zhetrd. <i>a</i> (<i>lda</i> ,*) is an array containing either upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> , and the strictly lower triangular part of <i>A</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i> , and the strictly upper triangular part of <i>A</i> is not referenced. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array (<i>lwork</i> \geq <i>n</i>). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	On exit, if <i>uplo</i> = 'U', the diagonal and first superdiagonal of <i>A</i> are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i> , and the elements above the first superdiagonal, with the array <i>tau</i> , represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors; if <i>uplo</i> = 'L', the diagonal and first subdiagonal of <i>A</i> are overwritten by the corresponding elements of the tridiagonal matrix <i>T</i> , and the elements below the first subdiagonal, with the array <i>tau</i> , represent the orthogonal matrix <i>Q</i> as a product of elementary reflectors.
<i>d, e</i>	REAL for chetrd DOUBLE PRECISION for zhetrd. Arrays: <i>d</i> (*) contains the diagonal elements of the matrix <i>T</i> . The dimension of <i>d</i> must be at least $\max(1, n)$.

	$e(*)$ contains the off-diagonal elements of T .
	The dimension of e must be at least $\max(1, n-1)$.
tau	COMPLEX for <code>chetrd</code> DOUBLE COMPLEX for <code>zhetrd</code> . Array, DIMENSION at least $\max(1, n-1)$. Stores further details of the unitary matrix Q .
$work(1)$	If $info = 0$, on exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hetrd` interface are the following:

a	Holds the matrix A of size (n,n) .
tau	Holds the vector of length $(n-1)$.
$uplo$	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Note that diagonal (d) and off-diagonal (e) elements of the matrix T are omitted because they are kept in the matrix A on exit.

Application Notes

For better performance, try using $lwork = n * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix T is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3) n^3$.

After calling this routine, you can call the following:

<code>ungtr</code>	to form the computed matrix Q explicitly
--------------------	--

unmtr to multiply a complex matrix by Q .

The real counterpart of this routine is [sytrd](#).

?ungtr

Generates the complex unitary matrix Q determined by [?hetrd](#).

Syntax

FORTRAN 77:

```
call cungtr(uplo, n, a, lda, tau, work, lwork, info)
call zungtr(uplo, n, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call ungtr(a, tau [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>ungtr( int matrix_layout, char uplo, lapack_int n, <datatype>* a,
lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine explicitly generates the n -by- n unitary matrix Q formed by [hetrd](#) when reducing a complex Hermitian matrix A to tridiagonal form: $A = Q^*T^*Q^H$. Use this routine after a call to [?hetrd](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?hetrd .
<i>n</i>	INTEGER. The order of the matrix Q ($n \geq 0$).
<i>a</i> , <i>tau</i> , <i>work</i>	COMPLEX for cungtr DOUBLE COMPLEX for zungtr. Arrays: <i>a</i> (<i>lda</i> ,*) is the array <i>a</i> as returned by ?hetrd . The second dimension of <i>a</i> must be at least max(1, <i>n</i>). <i>tau</i> (*) is the array <i>tau</i> as returned by ?hetrd . The dimension of <i>tau</i> must be at least max(1, <i>n</i> -1). <i>work</i> is a workspace array, its dimension max(1, <i>lwork</i>).

<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array (<i>lwork</i> $\geq n$). If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the unitary matrix <i>Q</i> .
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ungtr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n,n</i>).
<i>tau</i>	Holds the vector of length (<i>n-1</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

For better performance, try using *lwork* = (*n-1*) * *blocksize*, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

The real counterpart of this routine is [orgtr](#).

?unmtr

Multiples a complex matrix by the complex unitary matrix Q determined by ?hetrd.

Syntax**FORTRAN 77:**

```
call cunmtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
call zunmtr(side, uplo, trans, m, n, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmtr(a, tau, c [,side] [,uplo] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>unmtr( int matrix_layout, char side, char trans,
lapack_int m, lapack_int n, const <datatype>* a, lapack_int lda, const <datatype>*
tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine multiplies a complex matrix C by Q or Q^H , where Q is the unitary matrix Q formed by `hetrd` when reducing a complex Hermitian matrix A to tridiagonal form: $A = Q^*T^*Q^H$. Use this routine after a call to `?hetrd`.

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products Q^*C , Q^H*C , C^*Q , or C^*Q^H (overwriting the result on C).

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

In the descriptions below, r denotes the order of Q :

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

side CHARACTER*1. Must be either 'L' or 'R'.

If *side* = 'L', Q or Q^H is applied to C from the left.

If *side* = 'R', Q or Q^H is applied to C from the right.

uplo CHARACTER*1. Must be 'U' or 'L'.

Use the same *uplo* as supplied to `?hetrd`.

trans CHARACTER*1. Must be either 'N' or 'T'.

If *trans* = 'N', the routine multiplies C by Q .

If *trans* = 'T', the routine multiplies C by Q^H .

m INTEGER. The number of rows in the matrix C ($m \geq 0$).

<i>n</i>	INTEGER. The number of columns in <i>C</i> ($n \geq 0$).
<i>a, c, tau, work</i>	COMPLEX for <i>cunmtr</i> DOUBLE COMPLEX for <i>zunmtr</i> . <i>a</i> (<i>lda,*</i>) and <i>tau</i> are the arrays returned by <i>?hetrd</i> . The second dimension of <i>a</i> must be at least $\max(1, r)$. The dimension of <i>tau</i> must be at least $\max(1, r-1)$. <i>c</i> (<i>ldc,*</i>) contains the matrix <i>C</i> . The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work</i> is a workspace array, its dimension $\max(1, \text{lwork})$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $\text{lda} \geq \max(1, r)$.
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> ; $\text{ldc} \geq \max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array. Constraints: $\text{lwork} \geq \max(1, n)$ if <i>side</i> = 'L'; $\text{lwork} \geq \max(1, m)$ if <i>side</i> = 'R'. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , Q^H*C , C^*Q , or C^*Q^H (as specified by <i>side</i> and <i>trans</i>).
<i>work</i> (1)	If <i>info</i> = 0, on exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *unmtr* interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>r,r</i>). <i>r</i> = <i>m</i> if <i>side</i> = 'L'. <i>r</i> = <i>n</i> if <i>side</i> = 'R'.
<i>tau</i>	Holds the vector of length (<i>r-1</i>).

<i>c</i>	Holds the matrix <i>C</i> of size (<i>m,n</i>).
<i>side</i>	Must be ' <i>L</i> ' or ' <i>R</i> '. The default value is ' <i>L</i> '.
<i>uplo</i>	Must be ' <i>U</i> ' or ' <i>L</i> '. The default value is ' <i>U</i> '.
<i>trans</i>	Must be ' <i>N</i> ' or ' <i>C</i> '. The default value is ' <i>N</i> '.

Application Notes

For better performance, try using *lwork* = *n*blocksize* (for *side* = '*L*') or *lwork* = *m*blocksize* (for *side* = '*R*') where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed product differs from the exact product by a matrix *E* such that $\|E\|_2 = O(\epsilon) * \|C\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if *side* = '*L*' or $8*n^2*m$ if *side* = '*R*'.

The real counterpart of this routine is [ormtr](#).

?sptrd

Reduces a real symmetric matrix to tridiagonal form using packed storage.

Syntax

FORTRAN 77:

```
call ssprtd(uplo, n, ap, d, e, tau, info)
call dsprtd(uplo, n, ap, d, e, tau, info)
```

Fortran 95:

```
call sptrd(ap, tau [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>sprtd( int matrix_layout, char uplo, lapack_int n, <datatype>* ap, <datatype>* d, <datatype>* e, <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces a packed real symmetric matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q^T T Q$. The orthogonal matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation. See *Application Notes* below for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangle of A .
	If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangle of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap</i>	REAL for ssprtd DOUBLE PRECISION for dsprtd. Array, DIMENSION at least $\max(1, n(n+1)/2)$. Contains either upper or lower triangle of A (as specified by <i>uplo</i>) in the packed form described in "Matrix Arguments" in Appendix B .

Output Parameters

<i>ap</i>	Overwritten by the tridiagonal matrix T and details of the orthogonal matrix Q , as specified by <i>uplo</i> .
<i>d, e, tau</i>	REAL for ssprtd DOUBLE PRECISION for dsprtd. Arrays: <i>d</i> (*) contains the diagonal elements of the matrix T . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the off-diagonal elements of T . The dimension of <i>e</i> must be at least $\max(1, n-1)$. <i>tau</i> (*) stores further details of the matrix Q . The dimension of <i>tau</i> must be at least $\max(1, n-1)$.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sptrd` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>tau</code>	Holds the vector with the number of elements $n-1$.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Note that diagonal (`d`) and off-diagonal (`e`) elements of the matrix T are omitted because they are kept in the matrix A on exit.

Application Notes

The matrix Q is represented as a product of $n-1$ *elementary reflectors*, as follows :

- If `uplo = 'U'`, $Q = H(n-1) \dots H(2)H(1)$

Each $H(i)$ has the form

$H(i) = I - tau * v * v^T$ for real flavors, or

$H(i) = I - tau * v * v^H$ for complex flavors,

where tau is a real/complex scalar, and v is a real/complex vector with $v(i+1:n) = 0$ and $v(i) = 1$.

On exit, tau is stored in $tau(i)$, and $v(1:i-1)$ is stored in AP , overwriting $A(1:i-1, i+1)$.

- If `uplo = 'L'`, $Q = H(1)H(2) \dots H(n-1)$

Each $H(i)$ has the form

$H(i) = I - tau * v * v^T$ for real flavors, or

$H(i) = I - tau * v * v^H$ for complex flavors,

where tau is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$.

On exit, tau is stored in $tau(i)$, and $v(i+2:n)$ is stored in AP , overwriting $A(i+2:n, i)$.

The computed matrix T is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n)*\epsilon*\|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision. The approximate number of floating-point operations is $(4/3)n^3$.

After calling this routine, you can call the following:

`opgtr` to form the computed matrix Q explicitly

`opmtr` to multiply a real matrix by Q .

The complex counterpart of this routine is `hptrd`.

?opgtr

Generates the real orthogonal matrix Q determined by `?sptrd`.

Syntax

FORTRAN 77:

```
call sopgtr(uplo, n, ap, tau, q, ldq, work, info)
call dopgtr(uplo, n, ap, tau, q, ldq, work, info)
```

Fortran 95:

```
call opgtr(ap, tau, q [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>opgtr( int matrix_layout, char uplo, lapack_int n,
const <datatype>* ap, const <datatype>* tau, <datatype>* q, lapack_int ldq );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine explicitly generates the n -by- n orthogonal matrix Q formed by [?sptrd](#) when reducing a packed real symmetric matrix A to tridiagonal form: $A = Q^T T Q$. Use this routine after a call to [?sptrd](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be ' U ' or ' L '. Use the same <i>uplo</i> as supplied to ?sptrd .
<i>n</i>	INTEGER. The order of the matrix Q ($n \geq 0$).
<i>ap</i> , <i>tau</i>	REAL for sopgtr DOUBLE PRECISION for dopgtr . Arrays <i>ap</i> and <i>tau</i> , as returned by ?sptrd . The dimension of <i>ap</i> must be at least $\max(1, n(n+1)/2)$. The dimension of <i>tau</i> must be at least $\max(1, n-1)$.
<i>ldq</i>	INTEGER. The leading dimension of the output array <i>q</i> ; at least $\max(1, n)$.
<i>work</i>	REAL for sopgtr DOUBLE PRECISION for dopgtr . Workspace array, DIMENSION at least $\max(1, n-1)$.

Output Parameters

<i>q</i>	REAL for sopgtr DOUBLE PRECISION for dopgtr . Array, DIMENSION (<i>ldq</i> ,*). Contains the computed matrix Q . The second dimension of <i>q</i> must be at least $\max(1, n)$.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `opgtr` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>tau</code>	Holds the vector with the number of elements $n - 1$.
<code>q</code>	Holds the matrix Q of size (n,n) .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3)n^3$.

The complex counterpart of this routine is [upgtr](#).

?opmtr

Multiplies a real matrix by the real orthogonal matrix Q determined by ?sptrd.

Syntax

FORTRAN 77:

```
call sopmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call dopmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
```

Fortran 95:

```
call opmtr(ap, tau, c [,side] [,uplo] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>opmtr( int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const <datatype>* ap, const <datatype>* tau, <datatype>*
c, lapack_int ldc );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine multiplies a real matrix C by Q or Q^T , where Q is the orthogonal matrix Q formed by [sptrd](#) when reducing a packed real symmetric matrix A to tridiagonal form: $A = Q^T T^* Q^T$. Use this routine after a call to [sptrd](#).

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products $Q^* C$, $Q^T C$, $C^* Q$, or $C^* Q^T$ (overwriting the result on C).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

In the descriptions below, r denotes the order of Q :

If $\text{side} = \text{'L'}$, $r = m$; if $\text{side} = \text{'R'}$, $r = n$.

<i>side</i>	CHARACTER*1. Must be either 'L' or 'R'. If $\text{side} = \text{'L'}$, Q or Q^T is applied to C from the left. If $\text{side} = \text{'R'}$, Q or Q^T is applied to C from the right.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Use the same <i>uplo</i> as supplied to ?sptrd.
<i>trans</i>	CHARACTER*1. Must be either 'N' or 'T'. If $\text{trans} = \text{'N'}$, the routine multiplies C by Q . If $\text{trans} = \text{'T'}$, the routine multiplies C by Q^T .
<i>m</i>	INTEGER. The number of rows in the matrix C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>ap</i> , <i>tau</i> , <i>c</i> , <i>work</i>	REAL for sopmtr DOUBLE PRECISION for dopmtr. <i>ap</i> and <i>tau</i> are the arrays returned by ?sptrd. The dimension of <i>ap</i> must be at least $\max(1, r(r+1)/2)$. The dimension of <i>tau</i> must be at least $\max(1, r-1)$. <i>c</i> (<i>ldc</i> ,*) contains the matrix C . The second dimension of <i>c</i> must be at least $\max(1, n)$ <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$ if $\text{side} = \text{'L'}$; $\max(1, m)$ if $\text{side} = \text{'R'}$.
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> ; $\text{ldc} \geq \max(1, n)$.

Output Parameters

<i>c</i>	Overwritten by the product Q^*C , Q^T*C , C^*Q , or C^*Q^T (as specified by <i>side</i> and <i>trans</i>).
<i>info</i>	INTEGER. If $\text{info} = 0$, the execution is successful. If $\text{info} = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `opmtr` interface are the following:

<code>ap</code>	Holds the array A of size $(r*(r+1)/2)$, where $r = m$ if <code>side</code> = 'L'. $r = n$ if <code>side</code> = 'R'.
<code>tau</code>	Holds the vector with the number of elements $r - 1$.
<code>c</code>	Holds the matrix C of size (m,n) .
<code>side</code>	Must be 'L' or 'R'. The default value is 'L'.
<code>uplo</code>	Must be 'U' or 'L'. The default value is 'U'.
<code>trans</code>	Must be 'N', 'C', or 'T'. The default value is 'N'.

Application Notes

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\epsilon) \|C\|_2$, where ϵ is the machine precision.

The total number of floating-point operations is approximately $2*m^2*n$ if `side` = 'L', or $2*n^2*m$ if `side` = 'R'.

The complex counterpart of this routine is [upmtr](#).

?hptrd

Reduces a complex Hermitian matrix to tridiagonal form using packed storage.

Syntax

FORTRAN 77:

```
call chptrd(uplo, n, ap, d, e, tau, info)
call zhptrd(uplo, n, ap, d, e, tau, info)
```

Fortran 95:

```
call hptrd(ap, tau [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_chptrd( int matrix_layout, char uplo, lapack_int n,
                           lapack_complex_float* ap, float* d, float* e, lapack_complex_float* tau );
lapack_int LAPACKE_zhptrd( int matrix_layout, char uplo, lapack_int n,
                           lapack_complex_double* ap, double* d, double* e, lapack_complex_double* tau );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine reduces a packed complex Hermitian matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q^*T^*Q^H$. The unitary matrix Q is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with Q in this representation (see *Application Notes* below).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangle of A .
	If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangle of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap</i>	COMPLEX for chptrd DOUBLE COMPLEX for zhptrd. Array, DIMENSION at least max(1, $n(n+1)/2$). Contains either upper or lower triangle of A (as specified by <i>uplo</i>) in the packed form described in "Matrix Arguments" in Appendix B .

Output Parameters

<i>ap</i>	Overwritten by the tridiagonal matrix T and details of the orthogonal matrix Q , as specified by <i>uplo</i> .
<i>d, e</i>	REAL for chptrd DOUBLE PRECISION for zhptrd. Arrays: <i>d</i> (*) contains the diagonal elements of the matrix T . The dimension of <i>d</i> must be at least max(1, <i>n</i>). <i>e</i> (*) contains the off-diagonal elements of T . The dimension of <i>e</i> must be at least max(1, <i>n</i> -1).
<i>tau</i>	COMPLEX for chptrd DOUBLE COMPLEX for zhptrd. Arrays, DIMENSION at least max(1, <i>n</i> -1). Contains further details of the orthogonal matrix Q .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hptrd` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>tau</code>	Holds the vector with the number of elements $n - 1$.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Note that diagonal (`d`) and off-diagonal (`e`) elements of the matrix T are omitted because they are kept in the matrix A on exit.

Application Notes

The computed matrix T is exactly similar to a matrix $A + E$, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

After calling this routine, you can call the following:

<code>upgtr</code>	to form the computed matrix Q explicitly
<code>upmtr</code>	to multiply a complex matrix by Q .

The real counterpart of this routine is `sprtrd`.

?upgtr

Generates the complex unitary matrix Q determined by `?hptrd`.

Syntax

FORTRAN 77:

```
call cupgtr(uplo, n, ap, tau, q, ldq, work, info)
call zupgtr(uplo, n, ap, tau, q, ldq, work, info)
```

Fortran 95:

```
call upgtr(ap, tau, q [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>upgtr( int matrix_layout, char uplo, lapack_int n,
const <datatype>* ap, const <datatype>* tau, <datatype>* q, lapack_int ldq );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine explicitly generates the n -by- n unitary matrix Q formed by `hptrd` when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = Q*T*Q^H$. Use this routine after a call to `?hptrd`.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '. Use the same <i>uplo</i> as supplied to ?hptrd.
<i>n</i>	INTEGER. The order of the matrix <i>Q</i> (<i>n</i> \geq 0).
<i>ap</i> , <i>tau</i>	COMPLEX for cupgtr DOUBLE COMPLEX for zugptr. Arrays <i>ap</i> and <i>tau</i> , as returned by ?hptrd. The dimension of <i>ap</i> must be at least max(1, <i>n</i> (<i>n</i> +1)/2). The dimension of <i>tau</i> must be at least max(1, <i>n</i> -1).
<i>ldq</i>	INTEGER. The leading dimension of the output array <i>q</i> ; at least max(1, <i>n</i>).
<i>work</i>	COMPLEX for cupgtr DOUBLE COMPLEX for zugptr. Workspace array, DIMENSION at least max(1, <i>n</i> -1).

Output Parameters

<i>q</i>	COMPLEX for cupgtr DOUBLE COMPLEX for zugptr. Array, DIMENSION (<i>ldq</i> ,*). Contains the computed matrix <i>Q</i> . The second dimension of <i>q</i> must be at least max(1, <i>n</i>).
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *upgtr* interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size (<i>n</i> *(<i>n</i> +1)/2).
<i>tau</i>	Holds the vector with the number of elements <i>n</i> - 1.
<i>q</i>	Holds the matrix <i>Q</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be ' <i>U</i> ' or ' <i>L</i> '. The default value is ' <i>U</i> '.

Application Notes

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of floating-point operations is $(16/3)n^3$.

The real counterpart of this routine is [opgtr](#).

?upmtr

Multiples a complex matrix by the unitary matrix Q determined by ?hptrd.

Syntax

FORTRAN 77:

```
call cupmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
call zupmtr(side, uplo, trans, m, n, ap, tau, c, ldc, work, info)
```

Fortran 95:

```
call upmtr(ap, tau, c [,side] [,uplo] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>upmtr( int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const <datatype>* ap, const <datatype>* tau, <datatype>*
c, lapack_int ldc );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine multiplies a complex matrix C by Q or Q^H , where Q is the unitary matrix formed by [hptrd](#) when reducing a packed complex Hermitian matrix A to tridiagonal form: $A = Q*T*Q^H$. Use this routine after a call to [?hptrd](#).

Depending on the parameters *side* and *trans*, the routine can form one of the matrix products $Q*C$, Q^H*C , $C*Q$, or $C*Q^H$ (overwriting the result on C).

Input Parameters

The data types are given for the Fortran interface. A *<datatype>* placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

In the descriptions below, r denotes the order of Q :

If *side* = 'L', $r = m$; if *side* = 'R', $r = n$.

side CHARACTER*1. Must be either 'L' or 'R'.

If *side* = 'L', Q or Q^H is applied to C from the left.

If *side* = 'R', Q or Q^H is applied to C from the right.

uplo CHARACTER*1. Must be 'U' or 'L'.

Use the same *uplo* as supplied to ?hptrd.

trans

CHARACTER*1. Must be either 'N' or 'T'.

If *trans* = 'N', the routine multiplies C by Q .

If *trans* = 'T', the routine multiplies C by Q^H .

m

INTEGER. The number of rows in the matrix C ($m \geq 0$).

n

INTEGER. The number of columns in C ($n \geq 0$).

ap, tau, c,

COMPLEX for cupmtr

DOUBLE COMPLEX for zugmtr.

ap and *tau* are the arrays returned by ?hptrd.

The dimension of *ap* must be at least $\max(1, r(r+1)/2)$.

The dimension of *tau* must be at least $\max(1, r-1)$.

c(*ldc*,*) contains the matrix C .

The second dimension of *c* must be at least $\max(1, n)$

work(*) is a workspace array.

The dimension of *work* must be at least

$\max(1, n)$ if *side* = 'L';

$\max(1, m)$ if *side* = 'R'.

ldc

INTEGER. The leading dimension of *c*; *ldc* $\geq \max(1, n)$.

Output Parameters

c

Overwritten by the product Q^*C , Q^H*C , C^*Q , or C^*Q^H (as specified by *side* and *trans*).

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `upmtr` interface are the following:

ap

Holds the array A of size $(r*(r+1)/2)$, where

$r = m$ if *side* = 'L'.

$r = n$ if *side* = 'R'.

tau

Holds the vector with the number of elements $n - 1$.

c

Holds the matrix C of size (m,n) .

<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

The computed product differs from the exact product by a matrix E such that $\|E\|_2 = O(\varepsilon) * \|C\|_2$, where ε is the machine precision.

The total number of floating-point operations is approximately $8*m^2*n$ if *side* = 'L' or $8*n^2*m$ if *side* = 'R'.

The real counterpart of this routine is [opmtr](#).

?sbtrd

Reduces a real symmetric band matrix to tridiagonal form.

Syntax

FORTRAN 77:

```
call ssbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call dsbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
```

Fortran 95:

```
call sbtrd(ab[, q] [,vect] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>sbtrd( int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int kd, <datatype>* ab, lapack_int ldab, <datatype>* d, <datatype>* e,
<datatype>* q, lapack_int ldq );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces a real symmetric band matrix A to symmetric tridiagonal form T by an orthogonal similarity transformation: $A = Q*T*Q^T$. The orthogonal matrix Q is determined as a product of Givens rotations.

If required, the routine can also form the matrix Q explicitly.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>vect</i>	CHARACTER*1. Must be 'V' or 'N'.
	If <i>vect</i> = 'V', the routine returns the explicit matrix Q .

If $\text{vect} = \text{'N'}$, the routine does not return Q .

uplo CHARACTER*1. Must be ' U ' or ' L '.

If $\text{uplo} = \text{'U'}$, ab stores the upper triangular part of A .

If $\text{uplo} = \text{'L'}$, ab stores the lower triangular part of A .

n INTEGER. The order of the matrix A ($n \geq 0$).

kd INTEGER. The number of super- or sub-diagonals in A ($kd \geq 0$).

$ab, q, work$ REAL for ssbtrd

DOUBLE PRECISION for dsbtrd.

ab ($ldab, *$) is an array containing either upper or lower triangular part of the matrix A (as specified by uplo) in band storage format.

The second dimension of ab must be at least $\max(1, n)$.

q ($ldq, *$) is an array.

If $\text{vect} = \text{'U'}$, the q array must contain an n -by- n matrix X .

If $\text{vect} = \text{'N'}$ or ' V ', the q parameter need not be set.

The second dimension of q must be at least $\max(1, n)$.

$work(*)$ is a workspace array.

The dimension of $work$ must be at least $\max(1, n)$.

$ldab$ INTEGER. The leading dimension of ab ; at least $kd+1$.

ldq INTEGER. The leading dimension of q . Constraints:

$ldq \geq \max(1, n)$ if $\text{vect} = \text{'V'}$;

$ldq \geq 1$ if $\text{vect} = \text{'N'}$.

Output Parameters

ab

On exit, the diagonal elements of the array ab are overwritten by the diagonal elements of the tridiagonal matrix T . If $kd > 0$, the elements on the first superdiagonal (if $\text{uplo} = \text{'U'}$) or the first subdiagonal (if $\text{uplo} = \text{'L'}$) are overwritten by the off-diagonal elements of T . The rest of ab is overwritten by values generated during the reduction.

d, e, q

REAL for ssbtrd

DOUBLE PRECISION for dsbtrd.

Arrays:

$d(*)$ contains the diagonal elements of the matrix T .

The dimension of d must be at least $\max(1, n)$.

$e(*)$ contains the off-diagonal elements of T .

The dimension of e must be at least $\max(1, n-1)$.

$q(ldq, *)$ is not referenced if $\text{vect} = \text{'N'}$.

If `vect = 'V'`, q contains the n -by- n matrix Q .

The second dimension of q must be:

at least $\max(1, n)$ if `vect = 'V'`;

at least 1 if `vect = 'N'`.

`info` INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbtrd` interface are the following:

`ab` Holds the array A of size $(kd+1, n)$.

`q` Holds the matrix Q of size (n, n) .

`uplo` Must be '`U`' or '`L`'. The default value is '`U`'.

`vect` If omitted, this argument is restored based on the presence of argument `q` as follows: `vect = 'V'`, if `q` is present, `vect = 'N'`, if `q` is omitted.

If present, `vect` must be equal to '`V`' or '`U`' and the argument `q` must also be present. Note that there will be an error condition if `vect` is present and `q` omitted.

Note that diagonal (`d`) and off-diagonal (`e`) elements of the matrix T are omitted because they are kept in the matrix A on exit.

Application Notes

The computed matrix T is exactly similar to a matrix $A+E$, where $\|E\|_2 = c(n)*\varepsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ε is the machine precision. The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that $\|E\|_2 = O(\varepsilon)$.

The total number of floating-point operations is approximately $6n^2*kd$ if `vect = 'N'`, with $3n^3*(kd-1)/kd$ additional operations if `vect = 'V'`.

The complex counterpart of this routine is [hbtrd](#).

?hbtrd

Reduces a complex Hermitian band matrix to tridiagonal form.

Syntax

FORTRAN 77:

```
call chbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
call zhbtrd(vect, uplo, n, kd, ab, ldab, d, e, q, ldq, work, info)
```

Fortran 95:

```
call hbtrd(ab [, q] [, vect] [, uplo] [, info])
```

C:

```
lapack_int LAPACKE_chbtrd( int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int kd, lapack_complex_float* ab, lapack_int ldab, float* d, float* e,
lapack_complex_float* q, lapack_int ldq );
```

```
lapack_int LAPACKE_zhbtrd( int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int kd, lapack_complex_double* ab, lapack_int ldab, double* d, double* e,
lapack_complex_double* q, lapack_int ldq );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces a complex Hermitian band matrix A to symmetric tridiagonal form T by a unitary similarity transformation: $A = Q^*T^*Q^H$. The unitary matrix Q is determined as a product of Givens rotations.

If required, the routine can also form the matrix Q explicitly.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>vect</i>	CHARACTER*1. Must be 'V' or 'N'. If <i>vect</i> = 'V', the routine returns the explicit matrix Q . If <i>vect</i> = 'N', the routine does not return Q .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in A ($kd \geq 0$).
<i>ab</i> , <i>work</i>	COMPLEX for chbtrd DOUBLE COMPLEX for zhbtrd. <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the matrix A (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; at least $kd+1$.
<i>ldq</i>	INTEGER. The leading dimension of <i>q</i> . Constraints:

```

 $ldq \geq \max(1, n)$  if  $\text{vect} = 'V'$ ;  

 $ldq \geq 1$  if  $\text{vect} = 'N'$ .

```

Output Parameters

<i>ab</i>	On exit, the diagonal elements of the array <i>ab</i> are overwritten by the diagonal elements of the tridiagonal matrix <i>T</i> . If <i>kd</i> > 0, the elements on the first superdiagonal (if <i>uplo</i> = 'U') or the first subdiagonal (if <i>uplo</i> = 'L') are overwritten by the off-diagonal elements of <i>T</i> . The rest of <i>ab</i> is overwritten by values generated during the reduction.
<i>d, e</i>	<p>REAL for chbtrd</p> <p>DOUBLE PRECISION for zhbtrd.</p> <p>Arrays:</p> <p><i>d</i>(*) contains the diagonal elements of the matrix <i>T</i>.</p> <p>The dimension of <i>d</i> must be at least $\max(1, n)$.</p> <p><i>e</i>(*) contains the off-diagonal elements of <i>T</i>.</p> <p>The dimension of <i>e</i> must be at least $\max(1, n-1)$.</p>
<i>q</i>	<p>COMPLEX for chbtrd</p> <p>DOUBLE COMPLEX for zhbtrd.</p> <p>Array, DIMENSION (<i>ldq, *</i>).</p> <p>If $\text{vect} = 'N'$, <i>q</i> is not referenced.</p> <p>If $\text{vect} = 'V'$, <i>q</i> contains the <i>n</i>-by-<i>n</i> matrix <i>Q</i>.</p> <p>The second dimension of <i>q</i> must be:</p> <p>at least $\max(1, n)$ if $\text{vect} = 'V'$;</p> <p>at least 1 if $\text{vect} = 'N'$.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>th parameter had an illegal value.</p>

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbtrd` interface are the following:

<i>ab</i>	Holds the array <i>A</i> of size (<i>kd</i> +1, <i>n</i>).
<i>q</i>	Holds the matrix <i>Q</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vect</i>	If omitted, this argument is restored based on the presence of argument <i>q</i> as follows: <i>vect</i> = 'V', if <i>q</i> is present, <i>vect</i> = 'N', if <i>q</i> is omitted.

If present, *vect* must be equal to 'V' or 'U' and the argument *q* must also be present. Note that there will be an error condition if *vect* is present and *q* omitted.

Note that diagonal (*d*) and off-diagonal (*e*) elements of the matrix *T* are omitted because they are kept in the matrix *A* on exit.

Application Notes

The computed matrix *T* is exactly similar to a matrix *A + E*, where $\|E\|_2 = c(n) * \epsilon * \|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision. The computed matrix *Q* differs from an exactly unitary matrix by a matrix *E* such that $\|E\|_2 = O(\epsilon)$.

The total number of floating-point operations is approximately $20n^2 * kd$ if *vect* = 'N', with $10n^3 * (kd-1) / kd$ additional operations if *vect* = 'V'.

The real counterpart of this routine is [sbtrd](#).

?sterf

Computes all eigenvalues of a real symmetric tridiagonal matrix using QR algorithm.

Syntax

FORTRAN 77:

```
call ssterf(n, d, e, info)
call dsterf(n, d, e, info)
```

Fortran 95:

```
call sterf(d, e [,info])
```

C:

```
lapack_int LAPACKE_<?>sterf( lapack_int n, <datatype>* d, <datatype>* e );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues of a real symmetric tridiagonal matrix *T* (which can be obtained by reducing a symmetric or Hermitian matrix to tridiagonal form). The routine uses a square-root-free variant of the QR algorithm.

If you need not only the eigenvalues but also the eigenvectors, call [steqr](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

n INTEGER. The order of the matrix *T* ($n \geq 0$).

d, e REAL for ssterf

DOUBLE PRECISION for `dsterf`.

Arrays:

`d(*)` contains the diagonal elements of T .

The dimension of `d` must be at least $\max(1, n)$.

`e(*)` contains the off-diagonal elements of T .

The dimension of `e` must be at least $\max(1, n-1)$.

Output Parameters

`d`

The n eigenvalues in ascending order, unless `info > 0`.

See also `info`.

`e`

On exit, the array is overwritten; see `info`.

`info`

INTEGER.

If `info = 0`, the execution is successful.

If `info = i`, the algorithm failed to find all the eigenvalues after $30n$ iterations:

i off-diagonal elements have not converged to zero. On exit, `d` and `e` contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to T .

If `info = -i`, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sterf` interface are the following:

`d`

Holds the vector of length n .

`e`

Holds the vector of length $(n-1)$.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\varepsilon) * \|T\|_2$, where ε is the machine precision.

If λ_i is an exact eigenvalue, and m_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \varepsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about $14n^2$.

?steqr

Computes all eigenvalues and eigenvectors of a symmetric or Hermitian matrix reduced to tridiagonal form (QR algorithm).

Syntax

FORTRAN 77:

```
call ssteqr(compz, n, d, e, z, ldz, work, info)
call dsteqr(compz, n, d, e, z, ldz, work, info)
call csteqr(compz, n, d, e, z, ldz, work, info)
call zsteqr(compz, n, d, e, z, ldz, work, info)
```

Fortran 95:

```
call rsteqr(d, e [,z] [,compz] [,info])
call steqr(d, e [,z] [,compz] [,info])
```

C:

```
lapack_int LAPACKE_ssteqr( int matrix_layout, char compz, lapack_int n, float* d,
float* e, float* z, lapack_int ldz );
lapack_int LAPACKE_dsteqr( int matrix_layout, char compz, lapack_int n, double* d,
double* e, double* z, lapack_int ldz );
lapack_int LAPACKE_csteqr( int matrix_layout, char compz, lapack_int n, float* d,
float* e, lapack_complex_float* z, lapack_int ldz );
lapack_int LAPACKE_zsteqr( int matrix_layout, char compz, lapack_int n, double* d,
double* e, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric tridiagonal matrix T . In other words, the routine can compute the spectral factorization: $T = Z\Lambda Z^T$. Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i ; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$$T^*z_i = \lambda_i^*z_i \text{ for } i = 1, 2, \dots, n.$$

The routine normalizes the eigenvectors so that $\|z_i\|_2 = 1$.

You can also use the routine for computing the eigenvalues and eigenvectors of an arbitrary real symmetric (or complex Hermitian) matrix A reduced to tridiagonal form T : $A = Q^*T^*Q^H$. In this case, the spectral factorization is as follows: $A = Q^*T^*Q^H = (Q^*Z)^*\Lambda^*(Q^*Z)^H$. Before calling `?steqr`, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	?sytrd, ?orgtr	?hetrd, ?ungtr
packed storage	?sptrd, ?opgtr	?hptrd, ?upgtr
band storage	?sbtrd (<i>vect='V'</i>)	?hbtrd (<i>vect='V'</i>)

If you need eigenvalues only, it's more efficient to call `sterf`. If T is positive-definite, `pteqr` can compute small eigenvalues more accurately than `?steqr`.

To solve the problem by a single call, use one of the divide and conquer routines [stevd](#), [syevd](#), [spevd](#), or [sbevd](#) for real symmetric matrices or [heevd](#), [hpevd](#), or [hbevd](#) for complex Hermitian matrices.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>compz</i>	CHARACTER*1. Must be 'N' or 'I' or 'V'. If <i>compz</i> = 'N', the routine computes eigenvalues only. If <i>compz</i> = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T . If <i>compz</i> = 'V', the routine computes the eigenvalues and eigenvectors of A (and the array <i>z</i> must contain the matrix Q on entry).
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>d</i> , <i>e</i> , <i>work</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i> (*) contains the diagonal elements of T . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the off-diagonal elements of T . The dimension of <i>e</i> must be at least $\max(1, n-1)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be: at least 1 if <i>compz</i> = 'N'; at least $\max(1, 2*n-2)$ if <i>compz</i> = 'V' or 'I'.
<i>z</i>	REAL for <i>ssteqr</i> DOUBLE PRECISION for <i>dsteqr</i> COMPLEX for <i>csteqr</i> DOUBLE COMPLEX for <i>zsteqr</i> . Array, DIMENSION (<i>ldz</i> , *) If <i>compz</i> = 'N' or 'I', <i>z</i> need not be set. If <i>vect</i> = 'V', <i>z</i> must contain the n -by- n matrix Q . The second dimension of <i>z</i> must be: at least 1 if <i>compz</i> = 'N'; at least $\max(1, n)$ if <i>compz</i> = 'V' or 'I'. <i>work</i> (<i>lwork</i>) is a workspace array.
<i>ldz</i>	INTEGER. The leading dimension of <i>z</i> . Constraints: $ldz \geq 1$ if <i>compz</i> = 'N';

$ldz \geq \max(1, n)$ if $compz = 'V'$ or $'I'$.

Output Parameters

d	The n eigenvalues in ascending order, unless $info > 0$. See also $info$.
e	On exit, the array is overwritten; see $info$.
z	If $info = 0$, contains the n orthonormal eigenvectors, stored by columns. (The i -th column corresponds to the i th eigenvalue.)
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = i$, the algorithm failed to find all the eigenvalues after $30n$ iterations: i off-diagonal elements have not converged to zero. On exit, d and e contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to T . If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `steqr` interface are the following:

d	Holds the vector of length n .
e	Holds the vector of length $(n-1)$.
z	Holds the matrix Z of size (n,n) .
$compz$	If omitted, this argument is restored based on the presence of argument z as follows: $compz = 'I'$, if z is present, $compz = 'N'$, if z is omitted. If present, $compz$ must be equal to ' I ' or ' V ' and the argument z must also be present. Note that there will be an error condition if $compz$ is present and z omitted. Note that two variants of Fortran 95 interface for <code>steqr</code> routine are needed because of an ambiguous choice between real and complex cases appear when z is omitted. Thus, the name <code>rsteqr</code> is used in real cases (single or double precision), and the name <code>steqr</code> is used in complex cases (single or double precision).

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \epsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n) * \epsilon * \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.$$

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about

$24n^2$ if $compz = 'N'$;

$7n^3$ (for complex flavors, $14n^3$) if $compz = 'V'$ or $'I'$.

?stemr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)

call dstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)

call cstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)

call zstemr(jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, tryrac,
work, lwork, iwork, liwork, info)
```

C:

```
lapack_int LAPACKE_sstemr( int matrix_layout, char jobz, char range, lapack_int n,
const float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu,
lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int nzc, lapack_int* isuppz,
lapack_logical* tryrac );

lapack_int LAPACKE_dstemr( int matrix_layout, char jobz, char range, lapack_int n,
const double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu,
lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int nzc, lapack_int* isuppz,
lapack_logical* tryrac );

lapack_int LAPACKE_cstemr( int matrix_layout, char jobz, char range, lapack_int n,
const float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu,
lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz, lapack_int nzc,
lapack_int* isuppz, lapack_logical* tryrac );

lapack_int LAPACKE_zstemr( int matrix_layout, char jobz, char range, lapack_int n,
const double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu,
lapack_int* m, double* w, lapack_complex_double* z, lapack_int ldz, lapack_int nzc,
lapack_int* isuppz, lapack_logical* tryrac );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90

- C: mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T . Any such unreduced matrix has a well defined set of pairwise different real eigenvalues, the corresponding real eigenvectors are pairwise orthogonal.

The spectrum may be computed either completely or partially by specifying either an interval $(vl, vu]$ or a range of indices $il:iu$ for the desired eigenvalues.

Depending on the number of desired eigenvalues, these are computed either by bisection or the *dqds* algorithm. Numerically orthogonal eigenvectors are computed by the use of various suitable L^*D^*LT factorizations near clusters of close eigenvalues (referred to as RRRs, Relatively Robust Representations). An informal sketch of the algorithm follows.

For each unreduced block (submatrix) of T ,

- Compute $T - \sigma I = L^*D^*LT$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of L and D cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see steps c and d.
- For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- For each eigenvalue with a large enough relative separation compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to step c for any clusters that remain.

For more details, see: [\[Dhillon04\]](#), [\[Dhillon04-02\]](#), [\[Dhillon97\]](#)

The routine works only on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs (NaN stands for "not a number"). This permits the use of efficient inner loops avoiding a check for zero divisors.

LAPACK routines can be used to reduce a complex Hermitean matrix to real symmetric tridiagonal form.

(Any complex Hermitean tridiagonal matrix has real values on its diagonal and potentially complex numbers on its off-diagonals. By applying a similarity transform with an appropriate diagonal matrix $\text{diag}(1, e^{i\phi_1}, \dots, e^{i\phi_{n-1}})$, the complex Hermitean matrix can be transformed into a real symmetric matrix and complex arithmetic can be entirely avoided.) While the eigenvectors of the real symmetric tridiagonal matrix are real, the eigenvectors of original complex Hermitean matrix have complex entries in general. Since LAPACK drivers overwrite the matrix data with the eigenvectors, zstemr accepts complex workspace to facilitate interoperability with zunmtr or zugmtr.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues.

If $\text{range} = \text{'V'}$, the routine computes all eigenvalues in the half-open interval: $(\text{vl}, \text{vu}]$.

If $\text{range} = \text{'I'}$, the routine computes eigenvalues with indices il to iu .

n

INTEGER. The order of the matrix T ($n \geq 0$).

d

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (n).

Contains n diagonal elements of the tridiagonal matrix T .

e

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION ($n-1$).

Contains $(n-1)$ off-diagonal elements of the tridiagonal matrix T in elements 1 to $n-1$ of e . $e(n)$ need not be set on input, but is used internally as workspace.

vl, vu

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

If $\text{range} = \text{'V'}$, the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $\text{vl} < \text{vu}$.

If $\text{range} = \text{'A'}$ or 'I' , vl and vu are not referenced.

il, iu

INTEGER.

If $\text{range} = \text{'I'}$, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq \text{il} \leq \text{iu} \leq n$, if $n > 0$.

If $\text{range} = \text{'A'}$ or 'V' , il and iu are not referenced.

ldz

INTEGER. The leading dimension of the output array z .

if $\text{jobz} = \text{'V'}$, then $\text{ldz} \geq \max(1, n)$;

$\text{ldz} \geq 1$ otherwise.

nzc

INTEGER. The number of eigenvectors to be held in the array z .

If $\text{range} = \text{'A'}$, then $\text{nzc} \geq \max(1, n)$;

If $\text{range} = \text{'V'}$, then nzc is greater than or equal to the number of eigenvalues in the half-open interval: $(\text{vl}, \text{vu}]$.

If $\text{range} = \text{'I'}$, then $\text{nzc} \geq \text{iu} - \text{il} + 1$.

If $\text{nzc} = -1$, then a workspace query is assumed; the routine calculates the number of columns of the array z that are needed to hold the eigenvectors.

This value is returned as the first entry of the array z , and no error message related to nzc is issued by the routine `xerbla`.

tryrac

LOGICAL.

If `tryrac = .TRUE.`, it indicates that the code should check whether the tridiagonal matrix defines its eigenvalues to high relative accuracy. If so, the code uses relative-accuracy preserving algorithms that might be (a bit) slower depending on the matrix. If the matrix does not define its eigenvalues to high relative accuracy, the code can use possibly faster algorithms.

If `tryrac = .FALSE.`, the code is not required to guarantee relatively accurate eigenvalues and can use the fastest possible techniques.

`work` `REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Workspace array, DIMENSION (`lwork`).

`lwork` `INTEGER`.

The dimension of the array `work`,

`lwork ≥ max(1, 18*n)`.

If `lwork=-1`, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`.

`iwork` `INTEGER`.

Workspace array, DIMENSION (`liwork`).

`liwork` `INTEGER`.

The dimension of the array `iwork`.

`lwork≥max(1, 10*n)` if the eigenvectors are desired, and `lwork≥max(1, 8*n)` if only the eigenvalues are to be computed.

If `liwork=-1`, then a workspace query is assumed; the routine only calculates the optimal size of the `iwork` array, returns this value as the first entry of the `iwork` array, and no error message related to `liwork` is issued by `xerbla`.

Output Parameters

`d` On exit, the array `d` is overwritten.

`e` On exit, the array `e` is overwritten.

`m` `INTEGER`.

The total number of eigenvalues found, $0 \leq m \leq n$.

If `range = 'A'`, then $m=n$, and if If `range = 'I'`, then $m=iu-il+1$.

`w` `REAL` for single precision flavors

`DOUBLE PRECISION` for double precision flavors.

Array, DIMENSION (n).

The first m elements contain the selected eigenvalues in ascending order.

z REAL for sstemr
 DOUBLE PRECISION for dstemr
 COMPLEX for cstemr
 DOUBLE COMPLEX for zstemr.
 Array *z*(*ldz*, *), the second dimension of *z* must be at least $\max(1, m)$.
 If *jobz* = 'V', and *info* = 0, then the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w(i)*.
 If *jobz* = 'N', then *z* is not referenced.
 Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an can be computed with a workspace query by setting *nzc*=-1, see description of the parameter *nzc*.

isuppz INTEGER.
 Array, DIMENSION ($2 \times \max(1, m)$).
 The support of the eigenvectors in *z*, that is the indices indicating the nonzero elements in *z*. The *i*-th computed eigenvector is nonzero only in elements *isuppz*($2*i-1$) through *isuppz*($2*i$). This is relevant in the case when the matrix is split. *isuppz* is only accessed when *jobz* = 'V' and *n*>0.

tryrac On exit, TRUE. *tryrac* is set to .FALSE. if the matrix does not define its eigenvalues to high relative accuracy.

work(1) On exit, if *info* = 0, then *work(1)* returns the optimal (and minimal) size of *lwork*.

iwork(1) On exit, if *info* = 0, then *iwork(1)* returns the optimal size of *liwork*.

info INTEGER.
 If = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = 1, internal error in ?larre occurred,
 if *info* = 2, internal error in ?larry occurred.

?stedc

Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Syntax**FORTRAN 77:**

```
call sstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call dstedc(compz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
call cstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

```
call zstedc(compz, n, d, e, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call rstedc(d, e [,z] [,compz] [,info])
call stedc(d, e [,z] [,compz] [,info])
```

C:

```
lapack_int LAPACKE_sstedc( int matrix_layout, char compz, lapack_int n, float* d,
float* e, float* z, lapack_int ldz );
lapack_int LAPACKE_dstedc( int matrix_layout, char compz, lapack_int n, double* d,
double* e, double* z, lapack_int ldz );
lapack_int LAPACKE_cstedc( int matrix_layout, char compz, lapack_int n, float* d,
float* e, lapack_complex_float* z, lapack_int ldz );
lapack_int LAPACKE_zstedc( int matrix_layout, char compz, lapack_int n, double* d,
double* e, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The eigenvectors of a full or band real symmetric or complex Hermitian matrix can also be found if [sytrd/hetrd](#) or [sptrd/hptrd](#) or [sbtrd/hbtrd](#) has been used to reduce this matrix to tridiagonal form.

See also [laed0](#), [laed1](#), [laed2](#), [laed3](#), [laed4](#), [laed5](#), [laed6](#), [laed7](#), [laed8](#), [laed9](#), and [laeda](#) used by this function.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>compz</i>	CHARACTER*1. Must be 'N' or 'I' or 'V'. If <i>compz</i> = 'N', the routine computes eigenvalues only. If <i>compz</i> = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix. If <i>compz</i> = 'V', the routine computes the eigenvalues and eigenvectors of original symmetric/Hermitian matrix. On entry, the array <i>z</i> must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.
<i>n</i>	INTEGER. The order of the symmetric tridiagonal matrix (<i>n</i> \geq 0).
<i>d</i> , <i>e</i> , <i>rwork</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i> (*) contains the diagonal elements of the tridiagonal matrix.

The dimension of d must be at least $\max(1, n)$.

$e(*)$ contains the subdiagonal elements of the tridiagonal matrix.

The dimension of e must be at least $\max(1, n-1)$.

$rwork$ is a workspace array, its dimension $\max(1, lrwork)$.

$z, work$

REAL for sstedc

DOUBLE PRECISION for dstedc

COMPLEX for cstedc

DOUBLE COMPLEX for zstedc.

Arrays: $z(ldz, *)$, $work(*)$.

If $compz = 'V'$, then, on entry, z must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.

The second dimension of z must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

ldz

INTEGER. The leading dimension of z . Constraints:

$ldz \geq 1$ if $compz = 'N'$;

$ldz \geq \max(1, n)$ if $compz = 'V'$ or $'I'$.

$lwork$

INTEGER. The dimension of the array $work$.

For real functions sstedc and dstedc:

- If $compz = 'N'$ or $n \leq 1$, $lwork$ must be at least 1.
- If $compz = 'V'$ and $n > 1$, $lwork$ must be at least $1 + 3*n + 2*n*\log_2(n) + 4*n^2$, where $\log_2(n)$ is the smallest integer k such that $2^k \geq n$.
- If $compz = 'I'$ and $n > 1$ then $lwork$ must be at least $1 + 4*n + n^2$.

Note that for $compz = 'I'$ or $'V'$ and if n is less than or equal to the minimum divide size, usually 25, then $lwork$ need only be $\max(1, 2*(n-1))$.

For complex functions cstedc and zstedc:

- If $compz = 'N'$ or $'I'$, or $n \leq 1$, $lwork$ must be at least 1.
- If $compz = 'V'$ and $n > 1$, $lwork$ must be at least n^2 .

Note that for $compz = 'V'$, and if n is less than or equal to the minimum divide size, usually 25, then $lwork$ need only be 1.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See [Application Notes](#) for the required value of $lwork$.

$lrwork$

INTEGER. The dimension of the array $rwork$ (used for complex flavors only).

If $compz = 'N'$, or $n \leq 1$, $lrwork$ must be at least 1.

If $\text{compz} = \text{'V'}$ and $n > 1$, lrwork must be at least $(1 + 3*n + 2*n*\log_2(n) + 4*n^2)$, where $\log_2(n)$ is the smallest integer k such that $2^k \geq n$.

If $\text{compz} = \text{'I'}$ and $n > 1$, lrwork must be at least $(1 + 4*n + 2*n^2)$.

Note that for $\text{compz} = \text{'V'}$ or 'I' , and if n is less than or equal to the minimum divide size, usually 25, then lrwork need only be $\max(1, 2*(n-1))$.

If $\text{lrwork} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the work , rwork and iwork arrays, returns these values as the first entries of the work , rwork and iwork arrays, and no error message related to lwork or lrwork or liwork is issued by [xerbla](#). See *Application Notes* for the required value of lrwork .

iwork

INTEGER. Workspace array, its dimension $\max(1, \text{liwork})$.

liwork

INTEGER. The dimension of the array iwork .

If $\text{compz} = \text{'N'}$, or $n \leq 1$, liwork must be at least 1.

If $\text{compz} = \text{'V'}$ and $n > 1$, liwork must be at least $(6 + 6*n + 5*n*\log_2(n))$, where $\log_2(n)$ is the smallest integer k such that $2^k \geq n$.

If $\text{compz} = \text{'I'}$ and $n > 1$, liwork must be at least $(3 + 5*n)$.

Note that for $\text{compz} = \text{'V'}$ or 'I' , and if n is less than or equal to the minimum divide size, usually 25, then liwork need only be 1.

If $\text{liwork} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the work , rwork and iwork arrays, returns these values as the first entries of the work , rwork and iwork arrays, and no error message related to lwork or lrwork or liwork is issued by [xerbla](#). See *Application Notes* for the required value of liwork .

Output Parameters

d

The n eigenvalues in ascending order, unless $\text{info} \neq 0$.

See also info .

e

On exit, the array is overwritten; see info .

z

If $\text{info} = 0$, then if $\text{compz} = \text{'V'}$, z contains the orthonormal eigenvectors of the original symmetric/Hermitian matrix, and if $\text{compz} = \text{'I'}$, z contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If $\text{compz} = \text{'N'}$, z is not referenced.

$\text{work}(1)$

On exit, if $\text{info} = 0$, then $\text{work}(1)$ returns the optimal lwork .

$\text{rwork}(1)$

On exit, if $\text{info} = 0$, then $\text{rwork}(1)$ returns the optimal lrwork (for complex flavors only).

$\text{iwork}(1)$

On exit, if $\text{info} = 0$, then $\text{iwork}(1)$ returns the optimal liwork .

info

INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th parameter had an illegal value. If $\text{info} = i$, the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $i/(n+1)$ through $\text{mod}(i, n+1)$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stedc` interface are the following:

<i>d</i>	Holds the vector of length n .
<i>e</i>	Holds the vector of length $(n-1)$.
<i>z</i>	Holds the matrix Z of size (n,n) .
<i>compz</i>	If omitted, this argument is restored based on the presence of argument <i>z</i> as follows: $\text{compz} = 'I'$, if <i>z</i> is present, $\text{compz} = 'N'$, if <i>z</i> is omitted. If present, <i>compz</i> must be equal to ' <i>I</i> ' or ' <i>V</i> ' and the argument <i>z</i> must also be present. Note that there will be an error condition if <i>compz</i> is present and <i>z</i> omitted.

Note that two variants of Fortran 95 interface for `stedc` routine are needed because of an ambiguous choice between real and complex cases appear when *z* and *work* are omitted. Thus, the name `rstedc` is used in real cases (single or double precision), and the name `stedc` is used in complex cases (single or double precision).

Application Notes

The required size of workspace arrays must be as follows.

For `sstedc/dstedc`:

If $\text{compz} = 'N'$ or $n \leq 1$ then *lwork* must be at least 1.

If $\text{compz} = 'V'$ and $n > 1$ then *lwork* must be at least $(1 + 3n + 2n \cdot \log_2 n + 4n^2)$, where $\log_2(n) =$ smallest integer k such that $2^k \geq n$.

If $\text{compz} = 'I'$ and $n > 1$ then *lwork* must be at least $(1 + 4n + n^2)$.

If $\text{compz} = 'N'$ or $n \leq 1$ then *liwork* must be at least 1.

If $\text{compz} = 'V'$ and $n > 1$ then *liwork* must be at least $(6 + 6n + 5n \cdot \log_2 n)$.

If $\text{compz} = 'I'$ and $n > 1$ then *liwork* must be at least $(3 + 5n)$.

For `cstedc/zstedc`:

If $\text{compz} = 'N'$ or '*I*', or $n \leq 1$, *lwork* must be at least 1.

If $\text{compz} = 'V'$ and $n > 1$, *lwork* must be at least n^2 .

If $\text{compz} = 'N'$ or $n \leq 1$, *lrwork* must be at least 1.

If $\text{compz} = 'V'$ and $n > 1$, *lrwork* must be at least $(1 + 3n + 2n \cdot \log_2 n + 4n^2)$, where $\log_2(n) =$ smallest integer k such that $2^k \geq n$.

If $\text{compz} = 'I'$ and $n > 1$, *lrwork* must be at least $(1 + 4n + 2n^2)$.

The required value of *liwork* for complex flavors is the same as for real flavors.

If $lwork$ (or $liwork$ or $lrwork$, if supplied) is equal to -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$, $rwork$). This operation is called a workspace query.

Note that if $lwork$ ($liwork$, $lrwork$) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?stegr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

FORTRAN 77:

```
call ssteqr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)

call dstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)

call cstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)

call zstegr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)
```

Fortran 95:

```
call rstegr(d, e, w [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
call stegr(d, e, w [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

C:

```
lapack_int LAPACKE_ssteqr( int matrix_layout, char jobz, char range, lapack_int n,
float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* isuppz );

lapack_int LAPACKE_dstegr( int matrix_layout, char jobz, char range, lapack_int n,
double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu, double
abstol, lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* isuppz );

lapack_int LAPACKE_cstegr( int matrix_layout, char jobz, char range, lapack_int n,
float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz, lapack_int* isuppz );

lapack_int LAPACKE_zstegr( int matrix_layout, char jobz, char range, lapack_int n,
double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu, double
abstol, lapack_int* m, double* w, lapack_complex_double* z, lapack_int ldz,
lapack_int* isuppz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T . Any such unreduced matrix has a well defined set of pairwise different real eigenvalues, the corresponding real eigenvectors are pairwise orthogonal.

The spectrum may be computed either completely or partially by specifying either an interval $(vl, vu]$ or a range of indices $il:iu$ for the desired eigenvalues.

?sregr is a compatibility wrapper around the improved [stemr](#) routine. See its description for further details.

Note that the *abstol* parameter no longer provides any benefit and hence is no longer used.

See also auxiliary [lasq2](#) [lasq5](#), [lasq6](#), used by this routine.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>jobz</i>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>V</code> '. If <i>job</i> = ' <code>N</code> ', then only eigenvalues are computed. If <i>job</i> = ' <code>V</code> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <code>A</code> ' or ' <code>V</code> ' or ' <code>I</code> '. If <i>range</i> = ' <code>A</code> ', the routine computes all eigenvalues. If <i>range</i> = ' <code>V</code> ', the routine computes eigenvalues <i>lambda(i)</i> in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = ' <code>I</code> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>d, e, work</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays: <i>d(*)</i> contains the diagonal elements of T . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e(*)</i> contains the subdiagonal elements of T in elements 1 to $n-1$; <i>e(n)</i> need not be set on input, but it is used as a workspace. The dimension of <i>e</i> must be at least $\max(1, n)$. <i>work(lwork)</i> is a workspace array.
<i>vl, vu</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. If <i>range</i> = ' <code>V</code> ', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$.

If $\text{range} = \text{'A'}$ or 'I' , vl and vu are not referenced.

il, iu
INTEGER.
If $\text{range} = \text{'I'}$, the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$.
If $\text{range} = \text{'A'}$ or 'V' , il and iu are not referenced.

$abstol$
REAL for single precision flavors
DOUBLE PRECISION for double precision flavors.
Unused. Was the absolute error tolerance for the eigenvalues/eigenvectors in previous versions.

ldz
INTEGER. The leading dimension of the output array z . Constraints:
 $ldz < 1$ if $\text{jobz} = \text{'N'}$;
 $ldz < \max(1, n)$ $\text{jobz} = \text{'V'}$, an.

$lwork$
INTEGER.
The dimension of the array $work$,
 $lwork \geq \max(1, 18*n)$ if $\text{jobz} = \text{'V'}$, and
 $lwork \geq \max(1, 12*n)$ if $\text{jobz} = \text{'N'}$.
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#). See *Application Notes* below for details.

$iwork$
INTEGER.
Workspace array, size ($liwork$).

$liwork$
INTEGER.
The dimension of the array $iwork$, $liwork \geq \max(1, 10*n)$ if the eigenvectors are desired, and $liwork \geq \max(1, 8*n)$ if only the eigenvalues are to be computed..
If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $iwork$ array, returns this value as the first entry of the $iwork$ array, and no error message related to $liwork$ is issued by [xerbla](#). See *Application Notes* below for details.

Output Parameters

d, e
On exit, d and e are overwritten.

m
INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$.
If $\text{range} = \text{'A'}$, $m = n$, and if $\text{range} = \text{'I'}$, $m = iu - il + 1$.

w
REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.
 Array, size at least $\max(1, n)$.
 The selected eigenvalues in ascending order, stored in $w(1)$ to $w(m)$.

z REAL for sssteqr

DOUBLE PRECISION for dstegr

COMPLEX for cstegr

DOUBLE COMPLEX for zstegr.

Array $z(Idz, *)$, the second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, and if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If $jobz = 'N'$, then z is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used. Supplying n columns is always safe.

isuppz INTEGER.

Array, size at least $(2 * \max(1, m))$.

The support of the eigenvectors in z , that is the indices indicating the nonzero elements in z . The i -th computed eigenvector is nonzero only in elements $isuppz(2*i-1)$ through $isuppz(2*i)$. This is relevant in the case when the matrix is split. *isuppz* is only accessed when $jobz = 'V'$, and $n > 0$.

work(1)

On exit, if $info = 0$, then *work(1)* returns the required minimal size of *lwork*.

iwork(1)

On exit, if $info = 0$, then *iwork(1)* returns the required minimal size of *liwork*.

info INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = 1x$, internal error in ?larre occurred,

If $info = 2x$, internal error in ?larrv occurred. Here the digit $x = \text{abs}(iinfo) < 10$, where *iinfo* is the non-zero error code returned by ?larre or ?larrv, respectively.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *steqr* interface are the following:

<i>d</i>	Holds the vector of length <i>n</i> .
<i>e</i>	Holds the vector of length <i>n</i> .
<i>w</i>	Holds the vector of length <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n,m</i>).
<i>isuppz</i>	Holds the vector of length (2*m).
<i>vl</i>	Default value for this argument is <i>vl</i> = - HUGE (<i>vl</i>) where HUGE(<i>a</i>) means the largest machine number of the same precision as argument <i>a</i> .
<i>vu</i>	Default value for this argument is <i>vu</i> = HUGE (<i>vl</i>).
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this argument is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Note that two variants of Fortran 95 interface for `steqr` routine are needed because of an ambiguous choice between real and complex cases appear when *z* is omitted. Thus, the name `rstegr` is used in real cases (single or double precision), and the name `steqr` is used in complex cases (single or double precision).

Application Notes

Currently `?steqr` is only set up to find *all* the *n* eigenvalues and eigenvectors of *T* in $O(n^2)$ time, that is, only *range* = 'A' is supported.

`?steqr` works only on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs. Normal execution of `?steqr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not conform to the IEEE-754 standard.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?pteqr

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric positive-definite tridiagonal matrix.

Syntax

FORTRAN 77:

```
call spteqr(compz, n, d, e, z, ldz, work, info)
call dpteqr(compz, n, d, e, z, ldz, work, info)
call cpteqr(compz, n, d, e, z, ldz, work, info)
call zpteqr(compz, n, d, e, z, ldz, work, info)
```

Fortran 95:

```
call rpteqr(d, e [,z] [,compz] [,info])
call pteqr(d, e [,z] [,compz] [,info])
```

C:

```
lapack_int LAPACKE_sppeqr( int matrix_layout, char compz, lapack_int n, float* d,
float* e, float* z, lapack_int ldz );
lapack_int LAPACKE_dpteqr( int matrix_layout, char compz, lapack_int n, double* d,
double* e, double* z, lapack_int ldz );
lapack_int LAPACKE_cpteqr( int matrix_layout, char compz, lapack_int n, float* d,
float* e, lapack_complex_float* z, lapack_int ldz );
lapack_int LAPACKE_zpteqr( int matrix_layout, char compz, lapack_int n, double* d,
double* e, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric positive-definite tridiagonal matrix T . In other words, the routine can compute the spectral factorization: $T = Z^* \Lambda^* Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i ; Z is an orthogonal matrix whose columns are eigenvectors. Thus,

$$T^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

(The routine normalizes the eigenvectors so that $\|z_i\|_2 = 1$.)

You can also use the routine for computing the eigenvalues and eigenvectors of real symmetric (or complex Hermitian) positive-definite matrices A reduced to tridiagonal form T : $A = Q^* T^* Q^H$. In this case, the spectral factorization is as follows: $A = Q^* T^* Q^H = (QZ)^* \Lambda^* (QZ)^H$. Before calling ?pteqr, you must reduce A to tridiagonal form and generate the explicit matrix Q by calling the following routines:

	for real matrices:	for complex matrices:
full storage	?sytrd, ?orgtr	?hetrd, ?ungtr
packed storage	?sptrd, ?opgtr	?hptrd, ?upgtr
band storage	?sbtrd (<i>vect</i> ='V')	?hbtrd (<i>vect</i> ='V')

The routine first factorizes T as L^*D*L^H where L is a unit lower bidiagonal matrix, and D is a diagonal matrix. Then it forms the bidiagonal matrix $B = L^*D^{1/2}$ and calls `?bdsqr` to compute the singular values of B , which are the same as the eigenvalues of T .

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>compz</i>	CHARACTER*1. Must be 'N' or 'I' or 'V'. If <i>compz</i> = 'N', the routine computes eigenvalues only. If <i>compz</i> = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix T . If <i>compz</i> = 'V', the routine computes the eigenvalues and eigenvectors of A (and the array <i>z</i> must contain the matrix Q on entry).
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>d, e, work</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i> (*) contains the diagonal elements of T . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the off-diagonal elements of T . The dimension of <i>e</i> must be at least $\max(1, n-1)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be: at least 1 if <i>compz</i> = 'N'; at least $\max(1, 4*n-4)$ if <i>compz</i> = 'V' or 'I'.
<i>z</i>	REAL for <code>spteqr</code> DOUBLE PRECISION for <code>dpteqr</code> COMPLEX for <code>cpteqr</code> DOUBLE COMPLEX for <code>zpteqr</code> . Array, DIMENSION (<i>ldz</i> ,*) If <i>compz</i> = 'N' or 'I', <i>z</i> need not be set. If <i>compz</i> = 'V', <i>z</i> must contain the n -by- n matrix Q . The second dimension of <i>z</i> must be:

at least 1 if $\text{compz} = \text{'N'}$;
 at least $\max(1, n)$ if $\text{compz} = \text{'V'}$ or 'I' .

ldz INTEGER. The leading dimension of z . Constraints:
 $\text{ldz} \geq 1$ if $\text{compz} = \text{'N'}$;
 $\text{ldz} \geq \max(1, n)$ if $\text{compz} = \text{'V'}$ or 'I' .

Output Parameters

d The n eigenvalues in descending order, unless $\text{info} > 0$.
 See also info .

e On exit, the array is overwritten.

z If $\text{info} = 0$, contains the n orthonormal eigenvectors, stored by columns.
 (The i -th column corresponds to the i -th eigenvalue.)

info INTEGER.
 If $\text{info} = 0$, the execution is successful.
 If $\text{info} = i$, the leading minor of order i (and hence T itself) is not positive-definite.
 If $\text{info} = n + i$, the algorithm for computing singular values failed to converge; i off-diagonal elements have not converged to zero.
 If $\text{info} = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pteqr` interface are the following:

d Holds the vector of length n .

e Holds the vector of length $(n-1)$.

z Holds the matrix Z of size (n,n) .

compz If omitted, this argument is restored based on the presence of argument z as follows:
 $\text{compz} = \text{'I'}$, if z is present,
 $\text{compz} = \text{'N'}$, if z is omitted.
 If present, compz must be equal to 'I' or 'V' and the argument z must also be present. Note that there will be an error condition if compz is present and z omitted.

Note that two variants of Fortran 95 interface for `pteqr` routine are needed because of an ambiguous choice between real and complex cases appear when z is omitted. Thus, the name `rpteqr` is used in real cases (single or double precision), and the name `pteqr` is used in complex cases (single or double precision).

Application Notes

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \varepsilon * K * \lambda_i$$

where $c(n)$ is a modestly increasing function of n , ε is the machine precision, and $K = \|DTD\|_2 * \|D(DTD)^{-1}\|_2$, D is diagonal with $d_{ii} = t_{ii}^{-1/2}$.

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(u_i, w_i) \leq c(n)\varepsilon K / \min_{i \neq j}(|\lambda_i - \lambda_j| / |\lambda_i + \lambda_j|).$$

Here $\min_{i \neq j}(|\lambda_i - \lambda_j| / |\lambda_i + \lambda_j|)$ is the *relative gap* between λ_i and the other eigenvalues.

The total number of floating-point operations depends on how rapidly the algorithm converges.

Typically, it is about

$30n^2$ if $compz = 'N'$;

$6n^3$ (for complex flavors, $12n^3$) if $compz = 'V'$ or $'I'$.

?stebz

Computes selected eigenvalues of a real symmetric tridiagonal matrix by bisection.

Syntax

FORTRAN 77:

```
call sstebz (range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w, iblock,
isplit, work, iwork, info)

call dstebz (range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w, iblock,
isplit, work, iwork, info)
```

Fortran 95:

```
call stebz(d, e, m, nsplit, w, iblock, isplit [, order] [,vl] [,vu] [,il] [,iu]
[,abstol] [,info])
```

C:

```
lapack_int LAPACKE_<?>stebz( char range, char order, lapack_int n, <datatype> vl,
<datatype> vu, lapack_int il, lapack_int iu, <datatype> abstol, const <datatype>* d,
const <datatype>* e, lapack_int* m, lapack_int* nsplit, <datatype>* w, lapack_int*
iblock, lapack_int* isplit );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes some (or all) of the eigenvalues of a real symmetric tridiagonal matrix T by bisection. The routine searches for zero or negligible off-diagonal elements to see if T splits into block-diagonal form $T = \text{diag}(T_1, T_2, \dots)$. Then it performs bisection on each of the blocks T_i and returns the block index of each computed eigenvalue, so that a subsequent call to [stein](#) can also take advantage of the block structure.

See also [laebz](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues <i>lambda</i> (<i>i</i>) in the half-open interval: <i>vl</i> < <i>lambda</i> (<i>i</i>) ≤ <i>vu</i> . If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>order</i>	CHARACTER*1. Must be 'B' or 'E'. If <i>order</i> = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block. If <i>order</i> = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.
<i>n</i>	INTEGER. The order of the matrix <i>T</i> (<i>n</i> ≥ 0).
<i>vl</i> , <i>vu</i>	REAL for ssstebz DOUBLE PRECISION for dstebz. If <i>range</i> = 'V', the routine computes eigenvalues <i>lambda</i> (<i>i</i>) in the half-open interval: <i>vl</i> < <i>lambda</i> (<i>i</i>) ≤ <i>vu</i> . If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il</i> , <i>iu</i>	INTEGER. Constraint: 1 ≤ <i>il</i> ≤ <i>iu</i> ≤ <i>n</i> . If <i>range</i> = 'I', the routine computes eigenvalues <i>lambda</i> (<i>i</i>) such that <i>il</i> ≤ <i>i</i> ≤ <i>iu</i> (assuming that the eigenvalues <i>lambda</i> (<i>i</i>) are in ascending order). If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	REAL for ssstebz DOUBLE PRECISION for dstebz. The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width <i>abstol</i> . If <i>abstol</i> ≤ 0.0, then the tolerance is taken as <i>eps</i> * <i>T</i> , where <i>eps</i> is the machine precision, and <i>T</i> is the 1-norm of the matrix <i>T</i> .
<i>d</i> , <i>e</i> , <i>work</i>	REAL for ssstebz DOUBLE PRECISION for dstebz. Arrays: <i>d</i> (*) contains the diagonal elements of <i>T</i> . The dimension of <i>d</i> must be at least max(1, <i>n</i>). <i>e</i> (*) contains the off-diagonal elements of <i>T</i> .

The dimension of e must be at least $\max(1, n-1)$.
 $work(*)$ is a workspace array.
The dimension of $work$ must be at least $\max(1, 4n)$.

$iwork$ INTEGER. Workspace.

Array, DIMENSION at least $\max(1, 3n)$.

Output Parameters

m INTEGER. The actual number of eigenvalues found.

$nsplit$ INTEGER. The number of diagonal blocks detected in T .

w REAL for `sstebz`

DOUBLE PRECISION for `dstebz`.

Array, DIMENSION at least $\max(1, n)$. The computed eigenvalues, stored in $w(1)$ to $w(m)$.

$iblock, isplit$ INTEGER.

Arrays, DIMENSION at least $\max(1, n)$.

A positive value $iblock(i)$ is the block number of the eigenvalue stored in $w(i)$ (see also $info$).
The leading $nsplit$ elements of $isplit$ contain points at which T splits into blocks T_i as follows: the block T_1 contains rows/columns 1 to $isplit(1)$; the block T_2 contains rows/columns $isplit(1)+1$ to $isplit(2)$, and so on.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = 1$, for $range = 'A'$ or $'V'$, the algorithm failed to compute some of the required eigenvalues to the desired accuracy; $iblock(i) < 0$ indicates that the eigenvalue stored in $w(i)$ failed to converge.

If $info = 2$, for $range = 'I'$, the algorithm failed to compute some of the required eigenvalues. Try calling the routine again with $range = 'A'$.

If $info = 3$:

for $range = 'A'$ or $'V'$, same as $info = 1$;

for $range = 'I'$, same as $info = 2$.

If $info = 4$, no eigenvalues have been computed. The floating-point arithmetic on the computer is not behaving as expected.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stebz` interface are the following:

<i>d</i>	Holds the vector of length <i>n</i> .
<i>e</i>	Holds the vector of length (<i>n</i> -1).
<i>w</i>	Holds the vector of length <i>n</i> .
<i>iblock</i>	Holds the vector of length <i>n</i> .
<i>isplit</i>	Holds the vector of length <i>n</i> .
<i>order</i>	Must be 'B' or 'E'. The default value is 'B'.
<i>vl</i>	Default value for this argument is <i>vl</i> = - HUGE (<i>vl</i>) where HUGE(<i>a</i>) means the largest machine number of the same precision as argument <i>a</i> .
<i>vu</i>	Default value for this argument is <i>vu</i> = HUGE (<i>vl</i>).
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this argument is <i>abstol</i> = 0.0_WP.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

The eigenvalues of *T* are computed to high relative accuracy which means that if they vary widely in magnitude, then any small eigenvalues will be computed more accurately than, for example, with the standard QR method. However, the reduction to tridiagonal form (prior to calling the routine) may exclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix if its eigenvalues vary widely in magnitude.

?stein

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call dstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call cstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
call zstein(n, d, e, m, w, iblock, isplit, z, ldz, work, iwork, ifailv, info)
```

Fortran 95:

```
call stein(d, e, w, iblock, isplit, z [,ifailv] [,info])
```

C:

```

lapack_int LAPACKE_sstein( int matrix_layout, lapack_int n, const float* d,
                           const float* e, lapack_int m, const float* w, const lapack_int* iblock,
                           const lapack_int* isplit, float* z, lapack_int ldz, lapack_int* ifailv );
lapack_int LAPACKE_dstein( int matrix_layout, lapack_int n, const double* d,
                           const double* e, lapack_int m, const double* w, const lapack_int* iblock,
                           const lapack_int* isplit, double* z, lapack_int ldz, lapack_int* ifailv );
lapack_int LAPACKE_cstein( int matrix_layout, lapack_int n, const float* d,
                           const float* e, lapack_int m, const float* w, const lapack_int* iblock,
                           const lapack_int* isplit, lapack_complex_float* z, lapack_int ldz, lapack_int* ifailv );
lapack_int LAPACKE_zstein( int matrix_layout, lapack_int n, const double* d,
                           const double* e, lapack_int m, const double* w, const lapack_int* iblock,
                           const lapack_int* isplit, lapack_complex_double* z, lapack_int ldz, lapack_int* ifailv );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, by inverse iteration. It is designed to be used in particular after the specified eigenvalues have been computed by ?stebz with $order = 'B'$, but may also be used when the eigenvalues have been computed by other routines.

If you use this routine after ?stebz, it can take advantage of the block structure by performing inverse iteration on each block T_i separately, which is more efficient than using the whole matrix T .

If T has been formed by reduction of a full symmetric or Hermitian matrix A to tridiagonal form, you can transform eigenvectors of T to eigenvectors of A by calling ?ormtr or ?opmtr (for real flavors) or by calling ?unmtr or ?upmtr (for complex flavors).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>m</i>	INTEGER. The number of eigenvectors to be returned.
<i>d</i> , <i>e</i> , <i>w</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i> (*) contains the diagonal elements of T . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> (*) contains the sub-diagonal elements of T stored in elements 1 to $n-1$. The dimension of <i>e</i> must be at least $\max(1, n-1)$.

$w(*)$ contains the eigenvalues of T , stored in $w(1)$ to $w(m)$ (as returned by `stebz`). Eigenvalues of T_1 must be supplied first, in non-decreasing order; then those of T_2 , again in non-decreasing order, and so on. Constraint:

`if iblock(i) = iblock(i+1), w(i) ≤ w(i+1).`

The dimension of w must be at least $\max(1, n)$.

`iblock, isplit`

INTEGER.

Arrays, DIMENSION at least $\max(1, n)$. The arrays `iblock` and `isplit`, as returned by `?stebz` with `order = 'B'`.

If you did not call `?stebz` with `order = 'B'`, set all elements of `iblock` to 1, and `isplit(1)` to n .)

`ldz`

INTEGER. The leading dimension of the output array z ; $ldz \geq \max(1, n)$.

`work`

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Workspace array, DIMENSION at least $\max(1, 5n)$.

`iwork`

INTEGER.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

`z`

REAL for `sstein`

DOUBLE PRECISION for `dstein`

COMPLEX for `cstein`

DOUBLE COMPLEX for `zstein`.

Array, DIMENSION ($ldz, *$).

If `info = 0`, z contains the m orthonormal eigenvectors, stored by columns. (The i -th column corresponds to the i -th specified eigenvalue.)

`ifailv`

INTEGER.

Array, DIMENSION at least $\max(1, m)$.

If `info = i > 0`, the first i elements of `ifailv` contain the indices of any eigenvectors that failed to converge.

`info`

INTEGER.

If `info = 0`, the execution is successful.

If `info = i`, then i eigenvectors (as indicated by the parameter `ifailv`) each failed to converge in 5 iterations. The current iterates are stored in the corresponding columns of the array z .

If `info = -i`, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stein` interface are the following:

<code>d</code>	Holds the vector of length n .
<code>e</code>	Holds the vector of length n .
<code>w</code>	Holds the vector of length n .
<code>iblock</code>	Holds the vector of length n .
<code>isplit</code>	Holds the vector of length n .
<code>z</code>	Holds the matrix Z of size (n,m) .
<code>ifailv</code>	Holds the vector of length (m) .

Application Notes

Each computed eigenvector z_i is an exact eigenvector of a matrix $T+E_i$, where $\|E_i\|_2 = O(\epsilon) * \|T\|_2$. However, a set of eigenvectors computed by this routine may not be orthogonal to so high a degree of accuracy as those computed by `?steqr`.

?disna

Computes the reciprocal condition numbers for the eigenvectors of a symmetric/ Hermitian matrix or for the left or right singular vectors of a general matrix.

Syntax

FORTRAN 77:

```
call sdisna(job, m, n, d, sep, info)
call ddisna(job, m, n, d, sep, info)
```

Fortran 95:

```
call disna(d, sep [,job] [,minmn] [,info])
```

C:

```
lapack_int LAPACKE_<?>disna( char job, lapack_int m, lapack_int n, const <datatype>* d,
<datatype>* sep );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine computes the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general m -by- n matrix.

The reciprocal condition number is the 'gap' between the corresponding eigenvalue or singular value and the nearest other one.

The bound on the error, measured by angle in radians, in the i -th computed vector is given by

`slamch('E') * (anorm / sep(i))`

where $\text{anorm} = \|\mathbf{A}\|_2 = \max(|d(j)|)$. $\text{sep}(i)$ is not allowed to be smaller than `slamch('E') * anorm` in order to limit the size of the error bound.

?disna may also be used to compute error bounds for eigenvectors of the generalized symmetric definite eigenproblem.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

job CHARACTER*1. Must be 'E', 'L', or 'R'. Specifies for which problem the reciprocal condition numbers should be computed:

job = 'E': for the eigenvectors of a symmetric/Hermitian matrix;

job = 'L': for the left singular vectors of a general matrix;

job = 'R': for the right singular vectors of a general matrix.

m INTEGER. The number of rows of the matrix ($m \geq 0$).

n INTEGER.

If *job* = 'L', or 'R', the number of columns of the matrix ($n \geq 0$).
Ignored if *job* = 'E'.

d REAL for sdisna

DOUBLE PRECISION for ddisna.

Array, dimension at least $\max(1,m)$ if *job* = 'E', and at least $\max(1, \min(m,n))$ if *job* = 'L' or 'R'.

This array must contain the eigenvalues (if *job* = 'E') or singular values (if *job* = 'L' or 'R') of the matrix, in either increasing or decreasing order.

If singular values, they must be non-negative.

Output Parameters

sep REAL for sdisna

DOUBLE PRECISION for ddisna.

Array, dimension at least $\max(1,m)$ if *job* = 'E', and at least $\max(1, \min(m,n))$ if *job* = 'L' or 'R'. The reciprocal condition numbers of the vectors.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `disna` interface are the following:

<code>d</code>	Holds the vector of length $\min(m,n)$.
<code>sep</code>	Holds the vector of length $\min(m,n)$.
<code>job</code>	Must be 'E', 'L', or 'R'. The default value is 'E'.
<code>minmn</code>	Indicates which of the values m or n is smaller. Must be either 'M' or 'N', the default is 'M'. If <code>job</code> = 'E', this argument is superfluous, If <code>job</code> = 'L' or 'R', this argument is used by the routine.

Generalized Symmetric-Definite Eigenvalue Problems

Generalized symmetric-definite eigenvalue problems are as follows: find the eigenvalues λ and the corresponding eigenvectors z that satisfy one of these equations:

$$Az = \lambda Bz, ABz = \lambda z, \text{ or } BAz = \lambda z,$$

where A is an n -by- n symmetric or Hermitian matrix, and B is an n -by- n symmetric positive-definite or Hermitian positive-definite matrix.

In these problems, there exist n real eigenvectors corresponding to real eigenvalues (even for complex Hermitian matrices A and B).

Routines described in this section allow you to reduce the above generalized problems to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling LAPACK routines described earlier in this chapter (see [Symmetric Eigenvalue Problems](#)).

Different routines allow the matrices to be stored either conventionally or in packed storage. Prior to reduction, the positive-definite matrix B must first be factorized using either `potrf` or `pptrf`.

The reduction routine for the banded matrices A and B uses a split Cholesky factorization for which a specific routine `pbstf` is provided. This refinement halves the amount of work required to form matrix C .

Table "Computational Routines for Reducing Generalized Eigenproblems to Standard Problems" lists LAPACK routines (FORTRAN 77 interface) that can be used to solve generalized symmetric-definite eigenvalue problems. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Matrix type	Reduce to standard problems (full storage)	Reduce to standard problems (packed storage)	Reduce to standard problems (band matrices)	Factorize band matrix
real symmetric matrices	<code>sygst</code>	<code>spgst</code>	<code>sbgst</code>	<code>pbstf</code>
complex Hermitian matrices	<code>hegst</code>	<code>hpgst</code>	<code>hbgst</code>	<code>pbstf</code>

?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

FORTRAN 77:

```
call ssygst(itype, uplo, n, a, lda, b, ldb, info)
call dsygst(itype, uplo, n, a, lda, b, ldb, info)
```

Fortran 95:

```
call sygst(a, b [,itype] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>sygst( int matrix_layout, lapack_int itype, char uplo, lapack_int n, <datatype>* a, lapack_int lda, const <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces real symmetric-definite generalized eigenproblems

$$A^*z = \lambda * B^*z, \quad A^*B^*z = \lambda * z, \quad \text{or} \quad B^*A^*z = \lambda * z$$

to the standard form $C^*y = \lambda * y$. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call ?potrf to compute the Cholesky factorization: $B = U^T * U$ or $B = L * L^T$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. If <i>itype</i> = 1, the generalized eigenproblem is $A^*z = \lambda * B^*z$ for <i>uplo</i> = 'U': $C = \text{inv}(U^T) * A * \text{inv}(U)$, $z = \text{inv}(U) * y$; for <i>uplo</i> = 'L': $C = \text{inv}(L) * A * \text{inv}(L^T)$, $z = \text{inv}(L^T) * y$. If <i>itype</i> = 2, the generalized eigenproblem is $A^*B^*z = \lambda * z$ for <i>uplo</i> = 'U': $C = U * A * U^T$, $z = \text{inv}(U) * y$; for <i>uplo</i> = 'L': $C = L^T * A * L$, $z = \text{inv}(L^T) * y$. If <i>itype</i> = 3, the generalized eigenproblem is $B^*A^*z = \lambda * z$ for <i>uplo</i> = 'U': $C = U * A * U^T$, $z = U^T * y$; for <i>uplo</i> = 'L': $C = L^T * A * L$, $z = L * y$.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangle of <i>A</i> ; you must supply <i>B</i> in the factored form $B = U^T * U$. If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangle of <i>A</i> ; you must supply <i>B</i> in the factored form $B = L * L^T$.

<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>a, b</i>	REAL for ssygst DOUBLE PRECISION for dsygst. Arrays: <i>a</i> (<i>lda</i> ,*) contains the upper or lower triangle of <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb</i> ,*) contains the Cholesky-factored matrix <i>B</i> : $B = U^T * U$ or $B = L * L^T$ (as returned by ?potrf). The second dimension of <i>b</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	The upper or lower triangle of <i>A</i> is overwritten by the upper or lower triangle of <i>C</i> , as specified by the arguments <i>itype</i> and <i>uplo</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sygst` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n,n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n,n</i>).
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

Forming the reduced matrix *C* is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if *itype* = 1) or *B* (if *itype* = 2 or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if *B* is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?hegst

Reduces a complex Hermitian positive-definite generalized eigenvalue problem to the standard form.

Syntax

FORTRAN 77:

```
call chegst(itype, uplo, n, a, lda, b, ldb, info)
call zhegst(itype, uplo, n, a, lda, b, ldb, info)
```

Fortran 95:

```
call hegst(a, b [,itype] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_chegst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda, const lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zhegst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, const lapack_complex_double * b, lapack_int ldb);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces a complex Hermitian positive-definite generalized eigenvalue problem to standard form.

<i>itype</i>	Problem	Result
1	$A^*x = \lambda^*B^*x$	A overwritten by $\text{inv}(U^H)^*A^*\text{inv}(U)$ or $\text{inv}(L)^*A^*\text{inv}(L^H)$
2	$A^*B^*x = \lambda^*x$	A overwritten by $U^*A^*U^H$ or $L^H^*A^*L$
3	$B^*A^*x = \lambda^*x$	

Before calling this routine, you must call ?potrf to compute the Cholesky factorization: $B = U^H * U$ or $B = L * L^H$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. If <i>itype</i> = 1, the generalized eigenproblem is $A^*z = \lambda^*B^*z$ for <i>uplo</i> = 'U': $C = (U^H)^{-1} * A^*U^{-1}$; for <i>uplo</i> = 'L': $C = L^{-1} * A^*(L^H)^{-1}$. If <i>itype</i> = 2, the generalized eigenproblem is $A^*B^*z = \lambda^*z$ for <i>uplo</i> = 'U': $C = U^*A^*U^H$; for <i>uplo</i> = 'L': $C = L^H * A^*L$. If <i>itype</i> = 3, the generalized eigenproblem is $B^*A^*z = \lambda^*z$
--------------	---

```

for uplo = 'U': C = U*A*UH;
for uplo = 'L': C = LH*A*L.

uplo
CHARACTER*1. Must be 'U' or 'L'.

If uplo = 'U', the array a stores the upper triangle of A; you must supply
B in the factored form B = UH*U.

If uplo = 'L', the array a stores the lower triangle of A; you must supply
B in the factored form B = L*LH.

n
INTEGER. The order of the matrices A and B (n ≥ 0).

a, b
COMPLEX for chegst
DOUBLE COMPLEX for zhegst.

Arrays:
a(lda,*) contains the upper or lower triangle of A.
The second dimension of a must be at least max(1, n).
b ldb,*) contains the Cholesky-factored matrix B:
B = UH*U or B = L*LH (as returned by ?potrf).
The second dimension of b must be at least max(1, n).

lda
INTEGER. The leading dimension of a; at least max(1, n).

ldb
INTEGER. The leading dimension of b; at least max(1, n).

```

Output Parameters

a The upper or lower triangle of A is overwritten by the upper or lower triangle of C, as specified by the arguments *itype* and *uplo*.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hegst` interface are the following:

a	Holds the matrix A of size (n,n).
b	Holds the matrix B of size (n,n).
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by B^{-1} (if $itype = 1$) or B (if $itype = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?spgst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form using packed storage.

Syntax

FORTRAN 77:

```
call sspgst(itype, uplo, n, ap, bp, info)
call dspgst(itype, uplo, n, ap, bp, info)
```

Fortran 95:

```
call spgst(ap, bp [,itype] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>spgst( int matrix_layout, lapack_int itype, char uplo, lapack_int n, <datatype>* ap, const <datatype>* bp );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces real symmetric-definite generalized eigenproblems

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x$$

to the standard form $C^*y = \lambda^*y$, using packed matrix storage. Here A is a real symmetric matrix, and B is a real symmetric positive-definite matrix. Before calling this routine, call ?pptrf to compute the Cholesky factorization: $B = U^T * U$ or $B = L * L^T$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. If <i>itype</i> = 1, the generalized eigenproblem is $A^*z = \lambda^*B^*z$ for <i>uplo</i> = 'U': $C = \text{inv}(U^T) * A * \text{inv}(U)$, $z = \text{inv}(U) * y$; for <i>uplo</i> = 'L': $C = \text{inv}(L) * A * \text{inv}(L^T)$, $z = \text{inv}(L^T) * y$. If <i>itype</i> = 2, the generalized eigenproblem is $A^*B^*z = \lambda^*z$ for <i>uplo</i> = 'U': $C = U^T * A * U^T$, $z = \text{inv}(U) * y$;
--------------	---

```

for uplo = 'L': C = LT*A*L, z = inv(LT)*y.
If itype = 3, the generalized eigenproblem is B*A*z = lambda*z
for uplo = 'U': C = U*A*UT, z = UT*y;
for uplo = 'L': C = LT*A*L, z = L*y.

uplo
CHARACTER*1. Must be 'U' or 'L'.

If uplo = 'U', ap stores the packed upper triangle of A;
you must supply B in the factored form B = UT*U.

If uplo = 'L', ap stores the packed lower triangle of A;
you must supply B in the factored form B = L*LT.

n
INTEGER. The order of the matrices A and B (n ≥ 0).

ap, bp
REAL for sspgst
DOUBLE PRECISION for dspgst.

Arrays:
ap(*) contains the packed upper or lower triangle of A.
The dimension of ap must be at least max(1, n*(n+1)/2).
bp(*) contains the packed Cholesky factor of B (as returned by ?ptrf with
the same uplo value).
The dimension of bp must be at least max(1, n*(n+1)/2).

```

Output Parameters

ap The upper or lower triangle of A is overwritten by the upper or lower triangle of C, as specified by the arguments *itype* and *uplo*.

info INTEGER.
If *info* = 0, the execution is successful.
If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *spgst* interface are the following:

<i>ap</i>	Holds the array A of size (n*(n+1)/2).
<i>bp</i>	Holds the array B of size (n*(n+1)/2).
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if $\text{itype} = 1$) or B (if $\text{itype} = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?hpgst

Reduces a generalized eigenvalue problem with a Hermitian matrix to a standard eigenvalue problem using packed storage.

Syntax

FORTRAN 77:

```
call chpgst(itype, uplo, n, ap, bp, info)
call zhpgst(itype, uplo, n, ap, bp, info)
```

Fortran 95:

```
call hpgst(ap, bp [,itype] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>hpgst( int matrix_layout, lapack_int itype, char uplo, lapack_int n, <datatype>* ap, const <datatype>* bp );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces generalized eigenproblems with Hermitian matrices

$$A^*z = \lambda^*B^*z, A^*B^*z = \lambda^*z, \text{ or } B^*A^*z = \lambda^*z.$$

to standard eigenproblems $C^*y = \lambda^*y$, using packed matrix storage. Here A is a complex Hermitian matrix, and B is a complex Hermitian positive-definite matrix. Before calling this routine, you must call ?pptrf to compute the Cholesky factorization: $B = U^H * U$ or $B = L * L^H$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. If <i>itype</i> = 1, the generalized eigenproblem is $A^*z = \lambda^*B^*z$ for <i>uplo</i> = 'U': $C = \text{inv}(U^H) * A * \text{inv}(U)$, $z = \text{inv}(U) * y$; for <i>uplo</i> = 'L': $C = \text{inv}(L) * A * \text{inv}(L^H)$, $z = \text{inv}(L^H) * y$. If <i>itype</i> = 2, the generalized eigenproblem is $A^*B^*z = \lambda^*z$ for <i>uplo</i> = 'U': $C = U^*A^*U^H$, $z = \text{inv}(U) * y$;
--------------	---

```

for uplo = 'L': C = LH*A*L, z = inv(LH)*y.
If itype = 3, the generalized eigenproblem is B*A*z = lambda*z
for uplo = 'U': C = U*A*UH, z = UH*y;
for uplo = 'L': C = LH*A*L, z = L*y.

uplo
CHARACTER*1. Must be 'U' or 'L'.

If uplo = 'U', ap stores the packed upper triangle of A; you must supply
B in the factored form B = UH*U.

If uplo = 'L', ap stores the packed lower triangle of A; you must supply B
in the factored form B = L*LH.

n
INTEGER. The order of the matrices A and B (n ≥ 0).

ap, bp
COMPLEX for chpgstDOUBLE COMPLEX for zhpgst.

Arrays:

ap(*) contains the packed upper or lower triangle of A.
The dimension of a must be at least max(1, n*(n+1)/2).
bp(*) contains the packed Cholesky factor of B (as returned by ?pptrf with
the same uplo value).
The dimension of b must be at least max(1, n*(n+1)/2).

```

Output Parameters

ap The upper or lower triangle of A is overwritten by the upper or lower triangle of C, as specified by the arguments *itype* and *uplo*.

info INTEGER.

If info = 0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine hpgst interface are the following:

<i>ap</i>	Holds the array A of size (n*(n+1)/2).
<i>bp</i>	Holds the array B of size (n*(n+1)/2).
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

Forming the reduced matrix C is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if $\text{itype} = 1$) or B (if $\text{itype} = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is n^3 .

?sbgst

Reduces a real symmetric-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf.

Syntax

FORTRAN 77:

```
call ssbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, info)
call dsbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, info)
```

Fortran 95:

```
call sbgst(ab, bb [,x] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>sbgst( int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, <datatype>* ab, lapack_int ldab, const <datatype>* bb,
lapack_int ldbb, <datatype>* x, lapack_int ldx );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

To reduce the real symmetric-definite generalized eigenproblem $A^*z = \lambda^*B^*z$ to the standard form $C^*y = \lambda^*y$, where A , B and C are banded, this routine must be preceded by a call to **pbstf/pbstf**, which computes the split Cholesky factorization of the positive-definite matrix B : $B = S^T * S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites A with $C = X^T * A * X$, where $X = \text{inv}(S) * Q$ and Q is an orthogonal matrix chosen (implicitly) to preserve the bandwidth of A . The routine also has an option to allow the accumulation of X , and then, if z is an eigenvector of C , X^*z is an eigenvector of the original system.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

vect	CHARACTER*1. Must be 'N' or 'V'. If vect = 'N', then matrix X is not returned; If vect = 'V', then matrix X is returned.
uplo	CHARACTER*1. Must be 'U' or 'L'.

If $uplo = 'U'$, ab stores the upper triangular part of A .
If $uplo = 'L'$, ab stores the lower triangular part of A .

n INTEGER. The order of the matrices A and B ($n \geq 0$).
 ka INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).
 kb INTEGER. The number of super- or sub-diagonals in B ($ka \geq kb \geq 0$).
 $ab, bb, work$ REAL for ssbgst
DOUBLE PRECISION for dsbgst
 ab ($ldab, *$) is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by $uplo$) in band storage format.
The second dimension of the array ab must be at least $\max(1, n)$.
 bb ($ldb, *$) is an array containing the banded split Cholesky factor of B as specified by $uplo$, n and kb and returned by pbstf/pbstf.
The second dimension of the array bb must be at least $\max(1, n)$.
 $work(*)$ is a workspace array, dimension at least $\max(1, 2*n)$

$ldab$ INTEGER. The leading dimension of the array ab ; must be at least $ka+1$.
 ldb INTEGER. The leading dimension of the array bb ; must be at least $kb+1$.
 ldx The leading dimension of the output array x . Constraints: if $vect = 'N'$, then $ldx \geq 1$;
if $vect = 'V'$, then $ldx \geq \max(1, n)$.

Output Parameters

ab On exit, this array is overwritten by the upper or lower triangle of C as specified by $uplo$.
 x REAL for ssbgst
DOUBLE PRECISION for dsbgst
Array.
If $vect = 'V'$, then x ($ldx, *$) contains the n -by- n matrix $X = \text{inv}(S) * Q$.
If $vect = 'N'$, then x is not referenced.
The second dimension of x must be:
at least $\max(1, n)$, if $vect = 'V'$;
at least 1, if $vect = 'N'$.
 $info$ INTEGER.
If $info = 0$, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbgst` interface are the following:

<code>ab</code>	Holds the array A of size $(ka+1, n)$.
<code>bb</code>	Holds the array B of size $(kb+1, n)$.
<code>x</code>	Holds the matrix X of size (n, n) .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>vect</code>	Restored based on the presence of the argument <code>x</code> as follows: <code>vect = 'V'</code> , if <code>x</code> is present, <code>vect = 'N'</code> , if <code>x</code> is omitted.

Application Notes

Forming the reduced matrix C involves implicit multiplication by $\text{inv}(B)$. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion.

If ka and kb are much less than n then the total number of floating-point operations is approximately $6n^2 * kb$, when `vect = 'N'`. Additional $(3/2)n^3 * (kb/ka)$ operations are required when `vect = 'V'`.

?hbgst

Reduces a complex Hermitian positive-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf.

Syntax

FORTRAN 77:

```
call chbgst(vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, rwork, info)
call zhbgs(t vect, uplo, n, ka, kb, ab, ldab, bb, ldbb, x, ldx, work, rwork, info)
```

Fortran 95:

```
call hbgst(ab, bb [,x] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>hbgst( int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, <datatype>* ab, lapack_int ldab, const <datatype>* bb,
lapack_int ldbb, <datatype>* x, lapack_int ldx );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

To reduce the complex Hermitian positive-definite generalized eigenproblem $A^*z = \lambda^*B^*z$ to the standard form $C^*x = \lambda^*y$, where A , B and C are banded, this routine must be preceded by a call to [pbstf/pbstf](#), which computes the split Cholesky factorization of the positive-definite matrix B : $B = S^H S$. The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites A with $C = X^H A X$, where $X = \text{inv}(S) * Q$, and Q is a unitary matrix chosen (implicitly) to preserve the bandwidth of A . The routine also has an option to allow the accumulation of X , and then, if z is an eigenvector of C , X^*z is an eigenvector of the original system.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>vect</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>vect</i> = 'N', then matrix X is not returned; If <i>vect</i> = 'V', then matrix X is returned.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in B ($ka \geq kb \geq 0$).
<i>ab</i> , <i>bb</i> , <i>work</i>	COMPLEX for chbgstDOUBLE COMPLEX for zhbgst <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldbb</i> ,*) is an array containing the banded split Cholesky factor of B as specified by <i>uplo</i> , <i>n</i> and <i>kb</i> and returned by pbstf/pbstf . The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, dimension at least $\max(1, n)$
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>ldbb</i>	INTEGER. The leading dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>idx</i>	The leading dimension of the output array <i>x</i> . Constraints: if <i>vect</i> = 'N', then <i>idx</i> ≥ 1 ; if <i>vect</i> = 'V', then <i>idx</i> $\geq \max(1, n)$.
<i>rwork</i>	REAL for chbgst

DOUBLE PRECISION for zhbgst
 Workspace array, dimension at least max(1, n)

Output Parameters

<i>ab</i>	On exit, this array is overwritten by the upper or lower triangle of C as specified by <i>uplo</i> .
<i>x</i>	<p>COMPLEX for chbgst</p> <p>DOUBLE COMPLEX for zhbgst</p> <p>Array.</p> <p>If <i>vect</i> = 'V', then <i>x</i> (<i>idx</i>,*) contains the n-by-n matrix $X = \text{inv}(S) * Q$.</p> <p>If <i>vect</i> = 'N', then <i>x</i> is not referenced.</p> <p>The second dimension of <i>x</i> must be:</p> <p>at least max(1, n), if <i>vect</i> = 'V';</p> <p>at least 1, if <i>vect</i> = 'N'.</p>
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = $-i$, the <i>i</i>-th parameter had an illegal value.</p>

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbgst` interface are the following:

<i>ab</i>	Holds the array A of size $(ka+1, n)$.
<i>bb</i>	Holds the array B of size $(kb+1, n)$.
<i>x</i>	Holds the matrix X of size (n, n) .
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vect</i>	Restored based on the presence of the argument <i>x</i> as follows: <i>vect</i> = 'V', if <i>x</i> is present, <i>vect</i> = 'N', if <i>x</i> is omitted.

Application Notes

Forming the reduced matrix C involves implicit multiplication by $\text{inv}(B)$. When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if B is ill-conditioned with respect to inversion. The total number of floating-point operations is approximately $20n^2 * kb$, when *vect* = 'N'. Additional $5n^3 * (kb/ka)$ operations are required when *vect* = 'V'. All these estimates assume that both ka and kb are much less than n .

?pbstf

Computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite banded matrix used in ?sbgst/?hbgst .

Syntax**FORTRAN 77:**

```
call spbstf(uplo, n, kb, bb, ldbb, info)
call dpbstf(uplo, n, kb, bb, ldbb, info)
call cpbstf(uplo, n, kb, bb, ldbb, info)
call zpbstf(uplo, n, kb, bb, ldbb, info)
```

Fortran 95:

```
call pbstf(bb [, uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>pbstf( int matrix_layout, char uplo, lapack_int n, lapack_int kb,
<datatype>* bb, lapack_int ldbb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite band matrix B . It is to be used in conjunction with [sbgst/hbgst](#).

The factorization has the form $B = S^T * S$ (or $B = S^H * S$ for complex flavors), where S is a band matrix of the same bandwidth as B and the following structure: S is upper triangular in the first $(n+kb)/2$ rows and lower triangular in the remaining rows.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo	CHARACTER*1. Must be 'U' or 'L'.
	If $uplo = 'U'$, bb stores the upper triangular part of B .
	If $uplo = 'L'$, bb stores the lower triangular part of B .
n	INTEGER. The order of the matrix B ($n \geq 0$).
kb	INTEGER. The number of super- or sub-diagonals in B ($kb \geq 0$).
bb	REAL for spbstf DOUBLE PRECISION for dpbstf COMPLEX for cpbstf

DOUBLE COMPLEX for zpbstf.

bb (*lddb,**) is an array containing either upper or lower triangular part of the matrix *B* (as specified by *uplo*) in band storage format.

The second dimension of the array *bb* must be at least $\max(1, n)$.

lddb

INTEGER. The leading dimension of *bb*; must be at least *kb+1*.

Output Parameters

bb

On exit, this array is overwritten by the elements of the split Cholesky factor *S*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = *i*, then the factorization could not be completed, because the updated element b_{ii} would be the square root of a negative number; hence the matrix *B* is not positive-definite.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `pbstf` interface are the following:

bb Holds the array *B* of size (*kb+1,n*).

uplo Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed factor *S* is the exact factor of a perturbed matrix *B* + *E*, where

$$|E| \leq c(kb + 1)\varepsilon |S^H| |S|, \quad |e_{ij}| \leq c(kb + 1)\varepsilon \sqrt{b_{ii} b_{jj}}$$

c(n) is a modest linear function of *n*, and ε is the machine precision.

The total number of floating-point operations for real flavors is approximately $n(kb+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that *kb* is much less than *n*.

After calling this routine, you can call `sbgst/hgst` to solve the generalized eigenproblem $Az = \lambda Bz$, where *A* and *B* are banded and *B* is positive-definite.

Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

A *nonsymmetric eigenvalue problem* is as follows: given a nonsymmetric (or non-Hermitian) matrix *A*, find the *eigenvalues* λ and the corresponding *eigenvectors* *z* that satisfy the equation

$$Az = \lambda z \quad (\text{right eigenvectors } z)$$

or the equation

$$z^H A = \lambda z^H \text{ (left eigenvectors } z\text{).}$$

Nonsymmetric eigenvalue problems have the following properties:

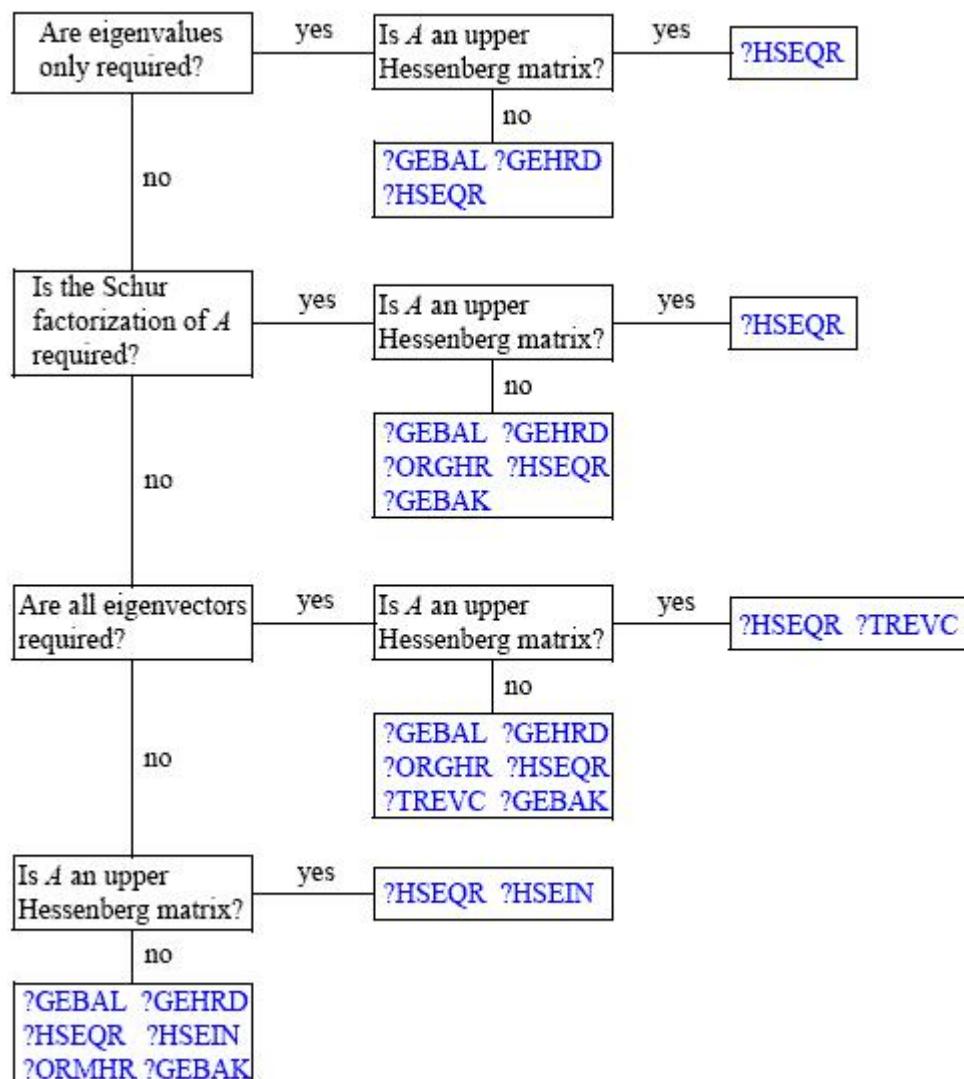
- The number of eigenvectors may be less than the matrix order (but is not less than the number of *distinct eigenvalues* of A).
- Eigenvalues may be complex even for a real matrix A .
- If a real nonsymmetric matrix has a complex eigenvalue $a+bi$ corresponding to an eigenvector z , then $a-bi$ is also an eigenvalue. The eigenvalue $a-bi$ corresponds to the eigenvector whose elements are complex conjugate to the elements of z .

To solve a nonsymmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained. [Table "Computational Routines for Solving Nonsymmetric Eigenvalue Problems"](#) lists LAPACK routines (FORTRAN 77 interface) to reduce the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$ as well as routines to solve eigenvalue problems with Hessenberg matrices, forming the Schur factorization of such matrices and computing the corresponding condition numbers. The corresponding routine names in the Fortran 95 interface are without the first symbol.

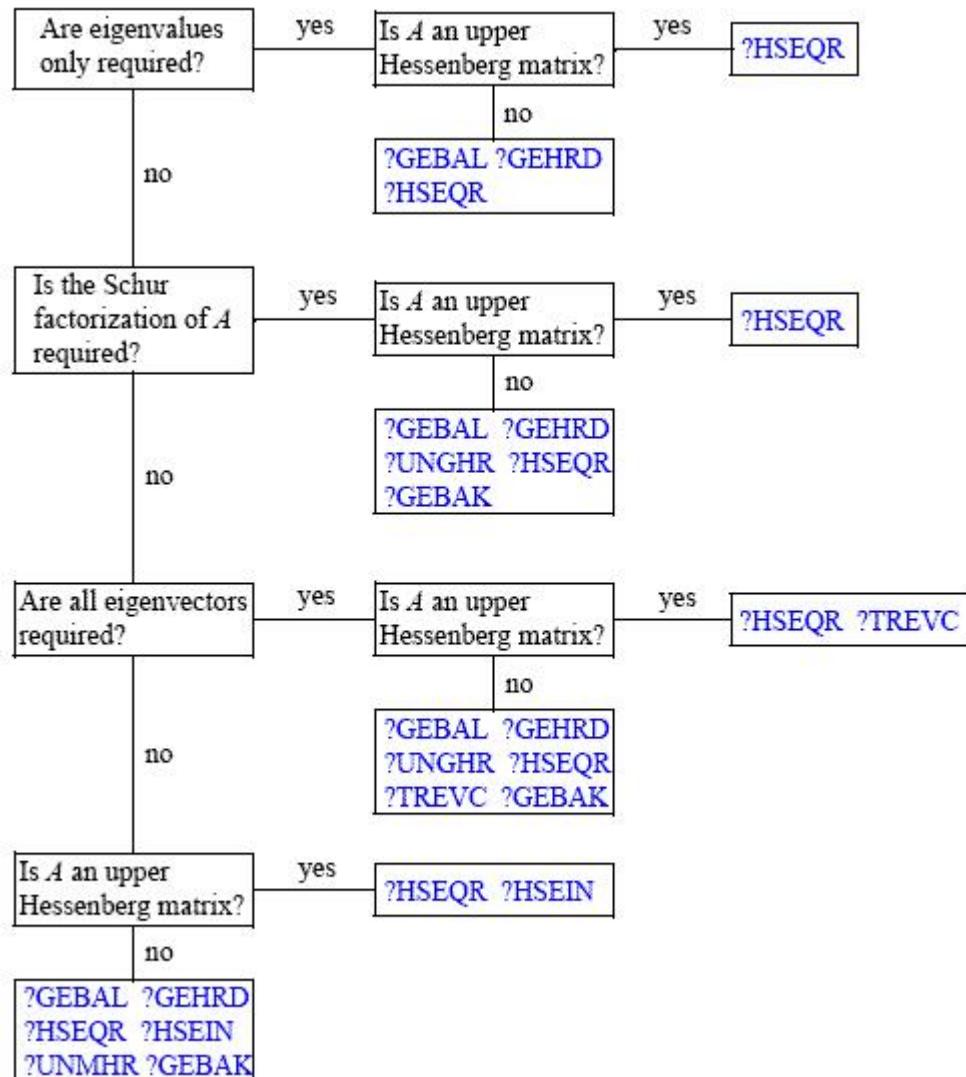
The decision tree in [Figure "Decision Tree: Real Nonsymmetric Eigenvalue Problems"](#) helps you choose the right routine or sequence of routines for an eigenvalue problem with a real nonsymmetric matrix. If you need to solve an eigenvalue problem with a complex non-Hermitian matrix, use the decision tree shown in [Figure "Decision Tree: Complex Non-Hermitian Eigenvalue Problems"](#).

[Computational Routines for Solving Nonsymmetric Eigenvalue Problems](#)

Operation performed	Routines for real matrices	Routines for complex matrices
Reduce to Hessenberg form $A = QHQ^H$?gehrd,	?gehrd
Generate the matrix Q	?orghr	?unghr
Apply the matrix Q	?ormhr	?unmhr
Balance matrix	?gebal	?gebal
Transform eigenvectors of balanced matrix to those of the original matrix	?gebak	?gebak
Find eigenvalues and Schur factorization (QR algorithm)	?hseqr	?hseqr
Find eigenvectors from Hessenberg form (inverse iteration)	?hsein	?hsein
Find eigenvectors from Schur factorization	?trevc	?trevc
Estimate sensitivities of eigenvalues and eigenvectors	?trsna	?trsna
Reorder Schur factorization	?trexc	?trexc
Reorder Schur factorization, find the invariant subspace and estimate sensitivities	?trsen	?trsen
Solves Sylvester's equation.	?trsyl	?trsyl

Decision Tree: Real Nonsymmetric Eigenvalue Problems

Decision Tree: Complex Non-Hermitian Eigenvalue Problems



?gehrd

Reduces a general matrix to upper Hessenberg form.

Syntax

FORTRAN 77:

```

call sgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call dgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call cgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
call zgehrd(n, ilo, ihi, a, lda, tau, work, lwork, info)
  
```

Fortran 95:

```
call gehrd(a [, tau] [,ilo] [,ihi] [,info])
```

C:

```
lapack_int LAPACKE_<?>gehrd( int matrix_layout, lapack_int n, lapack_int ilo,
lapack_int ihi, <datatype>* a, lapack_int lda, <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces a general matrix A to upper Hessenberg form H by an orthogonal or unitary similarity transformation $A = Q^*H^*Q^H$. Here H has real subdiagonal elements.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of *elementary reflectors*. Routines are provided to work with Q in this representation.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. If A is an output by ?gebal, then <i>ilo</i> and <i>ihi</i> must contain the values returned by that routine. Otherwise <i>ilo</i> = 1 and <i>ihi</i> = <i>n</i> . (If <i>n</i> > 0, then $1 \leq \text{ilo} \leq \text{ihi} \leq n$; if <i>n</i> = 0, <i>ilo</i> = 1 and <i>ihi</i> = 0.)
<i>a, work</i>	REAL for sgehrd DOUBLE PRECISION for dgehrd COMPLEX for cgehrd DOUBLE COMPLEX for zgehrd. Arrays: <i>a</i> (<i>lda,*</i>) contains the matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> (<i>lwork</i>) is a workspace array. <i>lda</i> INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The size of the <i>work</i> array; at least $\max(1, n)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	Overwritten by the upper Hessenberg matrix H and details of the matrix Q . The subdiagonal elements of H are real.
<i>tau</i>	REAL for sgehrd

DOUBLE PRECISION for dgehrd
 COMPLEX for cgehrd
 DOUBLE COMPLEX for zgehrd.
Array, DIMENSION at least max (1, $n-1$).
 Contains additional information on the matrix Q .

work(1)
 If $info = 0$, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info
 INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gehrd` interface are the following:

<i>a</i>	Holds the matrix A of size (n,n) .
<i>tau</i>	Holds the vector of length $(n-1)$.
<i>ilo</i>	Default value for this argument is $ilo = 1$.
<i>ihi</i>	Default value for this argument is $ihi = n$.

Application Notes

For better performance, try using $lwork = n * blocksize$, where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed Hessenberg matrix H is exactly similar to a nearby matrix $A + E$, where $\|E\|_2 < c(n)\epsilon\|A\|_2$, $c(n)$ is a modestly increasing function of n , and ϵ is the machine precision.

The approximate number of floating-point operations for real flavors is $(2/3)*(ihi - ilo)^2(2ihi + 2ilo + 3n)$; for complex flavors it is 4 times greater.

?orghr

Generates the real orthogonal matrix Q determined by ?gehrd.

Syntax**FORTRAN 77:**

```
call sorghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
call dorghr(n, ilo, ihi, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call orghr(a, tau [,ilo] [,ihi] [,info])
```

C:

```
lapack_int LAPACKE_<?>orghr( int matrix_layout, lapack_int n, lapack_int ilo,
lapack_int ihi, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine explicitly generates the orthogonal matrix Q that has been determined by a preceding call to sgehrd/dgehrd. (The routine ?gehrd reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q^*H^*Q^T$, and represents the matrix Q as a product of $ihi - ilo$ elementary reflectors. Here ilo and ihi are values determined by sgebal/dgebal when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

The matrix Q generated by ?orghr has the structure:

$$Q = \begin{bmatrix} I & 0 & 0 \\ 0 & Q_{22} & 0 \\ 0 & 0 & I \end{bmatrix}$$

where Q_{22} occupies rows and columns ilo to ihi .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of the matrix Q ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to ?gehrd. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, <i>ilo</i> = 1 and <i>ihi</i> = 0.)
<i>a, tau, work</i>	REAL for sorghr

DOUBLE PRECISION for `dorghr`

Arrays: $a(1:\text{lda},*)$ contains details of the vectors which define the elementary reflectors, as returned by `?gehrd`.

The second dimension of a must be at least $\max(1, n)$.

$\tau(1:\text{tau})$ contains further details of the elementary reflectors, as returned by `?gehrd`.

The dimension of τ must be at least $\max(1, n-1)$.

work is a workspace array, its dimension $\max(1, \text{lwork})$.

lda

INTEGER. The leading dimension of a ; at least $\max(1, n)$.

lwork

INTEGER. The size of the work array;

$\text{lwork} \geq \max(1, \text{ihi}-\text{ilo})$.

If $\text{lwork} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the work array, returns this value as the first entry of the work array, and no error message related to lwork is issued by `xerbla`.

See *Application Notes* for the suggested value of lwork .

Output Parameters

a

Overwritten by the n -by- n orthogonal matrix Q .

$\text{work}(1)$

If $\text{info} = 0$, on exit $\text{work}(1)$ contains the minimum value of lwork required for optimum performance. Use this lwork for subsequent runs.

info

INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `orghr` interface are the following:

a Holds the matrix A of size (n,n) .

τ Holds the vector of length $(n-1)$.

ilo Default value for this argument is $\text{ilo} = 1$.

ihi Default value for this argument is $\text{ihi} = n$.

Application Notes

For better performance, try using $\text{lwork} = (\text{ihi}-\text{ilo}) * \text{blocksize}$ where blocksize is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $\|E\|_2 = O(\epsilon)$, where ϵ is the machine precision.

The approximate number of floating-point operations is $(4/3)(ihi-ilo)^3$.

The complex counterpart of this routine is [unghr](#).

?ormhr

Multiples an arbitrary real matrix C by the real orthogonal matrix Q determined by ?gehrd.

Syntax

FORTRAN 77:

```
call sormhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
call dormhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call ormhr(a, tau, c [,ilo] [,ihi] [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>ormhr( int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int ilo, lapack_int ihi, const <datatype>* a, lapack_int lda,
const <datatype>* tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine multiplies a matrix *C* by the orthogonal matrix *Q* that has been determined by a preceding call to sgehrd/dgehrd. (The routine ?gehrd reduces a real general matrix *A* to upper Hessenberg form *H* by an orthogonal similarity transformation, $A = Q^*H^*Q^T$, and represents the matrix *Q* as a product of *ihi-ilo* elementary reflectors. Here *ilo* and *ihi* are values determined by sgebal/dgebal when balancing the matrix; if the matrix has not been balanced, *ilo* = 1 and *ihi* = *n*.)

With ?ormhr, you can form one of the matrix products Q^*C , Q^T*C , C^*Q , or C^*Q^T , overwriting the result on *C* (which may be any real rectangular matrix).

A common application of ?ormhr is to transform a matrix *V* of eigenvectors of *H* to the matrix *QV* of eigenvectors of *A*.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', then the routine forms Q^*C or Q^T*C . If <i>side</i> = 'R', then the routine forms C^*Q or C^*Q^T .
<i>trans</i>	CHARACTER*1. Must be 'N' or 'T'. If <i>trans</i> = 'N', then <i>Q</i> is applied to <i>C</i> . If <i>trans</i> = 'T', then Q^T is applied to <i>C</i> .
<i>m</i>	INTEGER. The number of rows in <i>C</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>C</i> ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to ?gehrd. If $m > 0$ and <i>side</i> = 'L', then $1 \leq ilo \leq ihi \leq m$. If $m = 0$ and <i>side</i> = 'L', then <i>ilo</i> = 1 and <i>ihi</i> = 0. If $n > 0$ and <i>side</i> = 'R', then $1 \leq ilo \leq ihi \leq n$. If $n = 0$ and <i>side</i> = 'R', then <i>ilo</i> = 1 and <i>ihi</i> = 0.
<i>a, tau, c, work</i>	REAL for sormhr DOUBLE PRECISION for dormhr Arrays: <i>a</i> (<i>lda,*</i>) contains details of the vectors which define the <i>elementary reflectors</i> , as returned by ?gehrd. The second dimension of <i>a</i> must be at least $\max(1, m)$ if <i>side</i> = 'L' and at least $\max(1, n)$ if <i>side</i> = 'R'. <i>tau</i> (*) contains further details of the <i>elementary reflectors</i> , as returned by ?gehrd . The dimension of <i>tau</i> must be at least $\max(1, m-1)$ if <i>side</i> = 'L' and at least $\max(1, n-1)$ if <i>side</i> = 'R'. <i>c</i> (<i>ldc,*</i>) contains the <i>m</i> by <i>n</i> matrix <i>C</i> . The second dimension of <i>c</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$. <i>lda</i> INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$ if <i>side</i> = 'L' and at least $\max(1, n)$ if <i>side</i> = 'R'. <i>ldc</i> INTEGER. The leading dimension of <i>c</i> ; at least $\max(1, m)$. <i>lwork</i> INTEGER. The size of the <i>work</i> array. If <i>side</i> = 'L', <i>lwork</i> $\geq \max(1, n)$.

If $\text{side} = \text{'R'}$, $\text{lwork} \geq \max(1, m)$.

If $\text{lwork} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the work array, returns this value as the first entry of the work array, and no error message related to lwork is issued by [xerbla](#).

See [Application Notes](#) for the suggested value of lwork .

Output Parameters

c	C is overwritten by product $Q^* C$, $Q^T * C$, $C * Q$, or $C * Q^T$ as specified by side and trans .
$\text{work}(1)$	If $\text{info} = 0$, on exit $\text{work}(1)$ contains the minimum value of lwork required for optimum performance. Use this lwork for subsequent runs.
info	INTEGER. If $\text{info} = 0$, the execution is successful. If $\text{info} = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ormhr` interface are the following:

a	Holds the matrix A of size (r,r) . $r = m$ if $\text{side} = \text{'L'}$. $r = n$ if $\text{side} = \text{'R'}$.
τ	Holds the vector of length $(r-1)$.
c	Holds the matrix C of size (m,n) .
ilo	Default value for this argument is $\text{ilo} = 1$.
ihi	Default value for this argument is $\text{ihi} = n$.
side	Must be 'L' or 'R' . The default value is 'L' .
trans	Must be 'N' or 'T' . The default value is 'N' .

Application Notes

For better performance, lwork should be at least $n*\text{blocksize}$ if $\text{side} = \text{'L'}$ and at least $m*\text{blocksize}$ if $\text{side} = \text{'R'}$, where blocksize is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of lwork for the first run or set $\text{lwork} = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $\|E\|_2 = O(\epsilon) * \|C\|_2$, where ϵ is the machine precision.

The approximate number of floating-point operations is

$2n(ihi-ilo)^2$ if *side* = 'L';

$2m(ihi-ilo)^2$ if *side* = 'R'.

The complex counterpart of this routine is [unmhr](#).

?unghr

Generates the complex unitary matrix *Q* determined by [?gehrd](#).

Syntax

FORTRAN 77:

```
call cunghr(n, ilo, ih, a, lda, tau, work, lwork, info)
call zunghr(n, ilo, ih, a, lda, tau, work, lwork, info)
```

Fortran 95:

```
call unghr(a, tau [,ilo] [,ih] [,info])
```

C:

```
lapack_int LAPACKE_<?>unghr( int matrix_layout, lapack_int n, lapack_int ilo,
lapack_int ih, <datatype>* a, lapack_int lda, const <datatype>* tau );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine is intended to be used following a call to [cgehrd/zgehrd](#), which reduces a complex matrix *A* to upper Hessenberg form *H* by a unitary similarity transformation: $A = Q^*H^*Q^H$. [?gehrd](#) represents the matrix *Q* as a product of *ih*-*ilo* elementary reflectors. Here *ilo* and *ih* are values determined by [cgebal/zgebal](#) when balancing the matrix; if the matrix has not been balanced, *ilo* = 1 and *ih* = *n*.

Use the routine [unghr](#) to generate *Q* explicitly as a square matrix. The matrix *Q* has the structure:

where Q_{22} occupies rows and columns $i\text{lo}$ to $i\text{hi}$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>n</i>	INTEGER. The order of the matrix Q ($n \geq 0$).
<i>i\text{lo}, i\text{hi}</i>	INTEGER. These must be the same parameters <i>i\text{lo}</i> and <i>i\text{hi}</i> , respectively, as supplied to ?gehrd . (If $n > 0$, then $1 \leq i\text{lo} \leq i\text{hi} \leq n$. If $n = 0$, then <i>i\text{lo} = 1</i> and <i>i\text{hi} = 0</i> .)
<i>a, tau, work</i>	COMPLEX for cunghr DOUBLE COMPLEX for zunghr. Arrays: <i>a</i> (<i>l\text{da}, *</i>) contains details of the vectors which define the <i>elementary reflectors</i> , as returned by ?gehrd. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>tau</i> (*) contains further details of the <i>elementary reflectors</i> , as returned by ?gehrd . The dimension of <i>tau</i> must be at least $\max(1, n-1)$. <i>work</i> is a workspace array, its dimension $\max(1, l\text{work})$. <i>l\text{da}</i> INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$. <i>l\text{work}</i> INTEGER. The size of the <i>work</i> array; $l\text{work} \geq \max(1, i\text{hi}-i\text{lo})$. If <i>l\text{work} = -1</i> , then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>l\text{work}</i> is issued by xerbla . See <i>Application Notes</i> for the suggested value of <i>l\text{work}</i> .

Output Parameters

<i>a</i>	Overwritten by the n -by- n unitary matrix Q .
<i>work</i> (1)	If <i>info = 0</i> , on exit <i>work</i> (1) contains the minimum value of <i>l\text{work}</i> required for optimum performance. Use this <i>l\text{work}</i> for subsequent runs.
<i>info</i>	INTEGER. If <i>info = 0</i> , the execution is successful. If <i>info = -i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmhr` interface are the following:

- `a` Holds the matrix A of size (n,n) .
- `tau` Holds the vector of length $(n-1)$.
- `ilo` Default value for this argument is $ilo = 1$.
- `ihi` Default value for this argument is $ihi = n$.

Application Notes

For better performance, try using $lwork = (ihi-ilo)*blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1 , then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix Q differs from the exact result by a matrix E such that $\|E\|_2 = O(\varepsilon)$, where ε is the machine precision.

The approximate number of real floating-point operations is $(16/3)(ihi-ilo)^3$.

The real counterpart of this routine is `orghr`.

?unmhr

Multiplies an arbitrary complex matrix C by the complex unitary matrix Q determined by ?gehrd.

Syntax

FORTRAN 77:

```
call cunmhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
call zunmhr(side, trans, m, n, ilo, ihi, a, lda, tau, c, ldc, work, lwork, info)
```

Fortran 95:

```
call unmhr(a, tau, c [,ilo] [,ihi] [,side] [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>unmhr( int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int ilo, lapack_int ihi, const <datatype>* a, lapack_int lda,
const <datatype>* tau, <datatype>* c, lapack_int ldc );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine multiplies a matrix C by the unitary matrix Q that has been determined by a preceding call to cgehrd/zgehrd. (The routine ?gehrd reduces a real general matrix A to upper Hessenberg form H by an orthogonal similarity transformation, $A = Q^H H Q^T$, and represents the matrix Q as a product of $ihi-ilo$ elementary reflectors. Here ilo and ihi are values determined by cgebal/zgebal when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

With ?unmhr, you can form one of the matrix products $Q^H C$, $Q^H C^H$, $C^H Q$, or $C^H Q^H$, overwriting the result on C (which may be any complex rectangular matrix). A common application of this routine is to transform a matrix V of eigenvectors of H to the matrix QV of eigenvectors of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', then the routine forms $Q^H C$ or $Q^H C^H$. If <i>side</i> = 'R', then the routine forms $C^H Q$ or $C^H Q^H$.
<i>trans</i>	CHARACTER*1. Must be 'N' or 'C'. If <i>trans</i> = 'N', then Q is applied to C . If <i>trans</i> = 'T', then Q^H is applied to C .
<i>m</i>	INTEGER. The number of rows in C ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in C ($n \geq 0$).
<i>ilo</i> , <i>ihi</i>	INTEGER. These must be the same parameters <i>ilo</i> and <i>ihi</i> , respectively, as supplied to ?gehrd . If $m > 0$ and <i>side</i> = 'L', then $1 \leq ilo \leq ihi \leq m$. If $m = 0$ and <i>side</i> = 'L', then <i>ilo</i> = 1 and <i>ihi</i> = 0. If $n > 0$ and <i>side</i> = 'R', then $1 \leq ilo \leq ihi \leq n$. If $n = 0$ and <i>side</i> = 'R', then <i>ilo</i> = 1 and <i>ihi</i> = 0.
<i>a</i> , <i>tau</i> , <i>c</i> , <i>work</i>	COMPLEX for cunmhr DOUBLE COMPLEX for zunmhr. Arrays: <i>a</i> (<i>lda</i> ,*) contains details of the vectors which define the elementary reflectors, as returned by ?gehrd. The second dimension of <i>a</i> must be at least $\max(1, m)$ if <i>side</i> = 'L' and at least $\max(1, n)$ if <i>side</i> = 'R'.

tau(*) contains further details of the elementary reflectors, as returned by ?gehrd.

The dimension of *tau* must be at least max (1, *m*-1)

if *side* = 'L' and at least max (1, *n*-1) if *side* = 'R'.

c (*lrc*,*) contains the *m*-by-*n* matrix *C*. The second dimension of *c* must be at least max(1, *n*).

work is a workspace array, its dimension max(1, *lwork*).

lda INTEGER. The leading dimension of *a*; at least max(1, *m*) if *side* = 'L' and at least max (1, *n*) if *side* = 'R'.

ldc INTEGER. The leading dimension of *c*; at least max(1, *m*).

lwork INTEGER. The size of the *work* array.

If *side* = 'L', *lwork* $\geq \max(1, n)$.

If *side* = 'R', *lwork* $\geq \max(1, m)$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

c *C* is overwritten by *Q***C*, or *Q*^H**C*, or *C***Q*^H, or *C***Q* as specified by *side* and *trans*.

work(1) If *info* = 0, on exit *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `unmhr` interface are the following:

a Holds the matrix *A* of size (*r*,*r*).

r = *m* if *side* = 'L'.

r = *n* if *side* = 'R'.

tau Holds the vector of length (*r*-1).

c Holds the matrix *C* of size (*m*,*n*).

<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>side</i>	Must be 'L' or 'R'. The default value is 'L'.
<i>trans</i>	Must be 'N' or 'C'. The default value is 'N'.

Application Notes

For better performance, *lwork* should be at least $n * \text{blocksize}$ if *side* = 'L' and at least $m * \text{blocksize}$ if *side* = 'R', where *blocksize* is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The computed matrix *Q* differs from the exact result by a matrix *E* such that $\|E\|_2 = O(\epsilon) * \|C\|_2$, where ϵ is the machine precision.

The approximate number of floating-point operations is

$8n(ihi-ilo)^2$ if *side* = 'L';

$8m(ihi-ilo)^2$ if *side* = 'R'.

The real counterpart of this routine is [ormhr](#).

?gebal

Balances a general matrix to improve the accuracy of computed eigenvalues and eigenvectors.

Syntax

FORTRAN 77:

```
call sgebal(job, n, a, lda, ilo, ihi, scale, info)
call dgebal(job, n, a, lda, ilo, ihi, scale, info)
call cgebal(job, n, a, lda, ilo, ihi, scale, info)
call zgebal(job, n, a, lda, ilo, ihi, scale, info)
```

Fortran 95:

```
call gebal(a [, scale] [,ilo] [,ihi] [,job] [,info])
```

C:

```
lapack_int LAPACKE_sgebal( int matrix_layout, char job, lapack_int n, float* a,
                           lapack_int lda, lapack_int* ilo, lapack_int* ihi, float* scale );
lapack_int LAPACKE_dgebal( int matrix_layout, char job, lapack_int n, double* a,
                           lapack_int lda, lapack_int* ilo, lapack_int* ihi, double* scale );
```

```
lapack_int LAPACKE_cgebal( int matrix_layout, char job, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_int* ilo, lapack_int* ihi, float*
scale );

lapack_int LAPACKE_zgebal( int matrix_layout, char job, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_int* ilo, lapack_int* ihi, double*
scale );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine *balances* a matrix A by performing either or both of the following two similarity transformations:

(1) The routine first attempts to permute A to block upper triangular form:

$$\dots \begin{array}{|c|c|} \hline 1 & * \\ \hline \end{array} \dots \begin{array}{|c|c|} \hline 1 & * \\ \hline \end{array} \dots \begin{array}{|c|c|} \hline 1 & * \\ \hline \end{array}$$

where P is a permutation matrix, and A'_{11} and A'_{33} are upper triangular. The diagonal elements of A'_{11} and A'_{33} are eigenvalues of A . The rest of the eigenvalues of A are the eigenvalues of the central diagonal block A'_{22} , in rows and columns ilo to ihi . Subsequent operations to compute the eigenvalues of A (or its Schur factorization) need only be applied to these rows and columns; this can save a significant amount of work if $ilo > 1$ and $ihi < n$.

If no suitable permutation exists (as is often the case), the routine sets $ilo = 1$ and $ihi = n$, and A'_{22} is the whole of A .

(2) The routine applies a diagonal similarity transformation to A' , to make the rows and columns of A'_{22} as close in norm as possible:

$$\dots \begin{array}{|c|c|} \hline 1 & * \\ \hline \end{array} \dots \begin{array}{|c|c|} \hline 1 & * \\ \hline \end{array} \dots \begin{array}{|c|c|} \hline 1 & * \\ \hline \end{array} \dots$$

This scaling can reduce the norm of the matrix (that is, $\|A'_{22}\| < \|A\|\$), and hence reduce the effect of rounding errors on the accuracy of computed eigenvalues and eigenvectors.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>job</i>	CHARACTER*1. Must be 'N' or 'P' or 'S' or 'B'.
------------	--

If $job = 'N'$, then A is neither permuted nor scaled (but ilo , ih_i , and $scale$ get their values).

If $job = 'P'$, then A is permuted but not scaled.

If $job = 'S'$, then A is scaled but not permuted.

If $job = 'B'$, then A is both scaled and permuted.

n INTEGER. The order of the matrix A ($n \geq 0$).

a REAL for sgebal
 DOUBLE PRECISION for dgebal
 COMPLEX for cgebal
 DOUBLE COMPLEX for zgebal.

Arrays:

a ($lda,*$) contains the matrix A .

The second dimension of *a* must be at least $\max(1, n)$. *a* is not referenced if $job = 'N'$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

Output Parameters

a Overwritten by the balanced matrix (*a* is not referenced if $job = 'N'$).

ilo, *ih_i* INTEGER. The values *ilo* and *ih_i* such that on exit $a(i,j)$ is zero if $i > j$ and $1 \leq j < ilo$ or $ih_i < i \leq n$.
 If $job = 'N'$ or $'S'$, then $ilo = 1$ and $ih_i = n$.

scale REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors
 Array, DIMENSION at least $\max(1, n)$.

Contains details of the permutations and scaling factors.

More precisely, if p_j is the index of the row and column interchanged with row and column j , and d_j is the scaling factor used to balance row and column j , then

$scale(j) = p_j$ for $j = 1, 2, \dots, ilo-1, ih_i+1, \dots, n$;
 $scale(j) = d_j$ for $j = ilo, ilo + 1, \dots, ih_i$.

The order in which the interchanges are made is n to ih_i+1 , then 1 to $ilo-1$.

info INTEGER.
 If $info = 0$, the execution is successful.
 If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gebal` interface are the following:

<code>a</code>	Holds the matrix A of size (n,n) .
<code>scale</code>	Holds the vector of length n .
<code>ilo</code>	Default value for this argument is <code>ilo = 1</code> .
<code>ihi</code>	Default value for this argument is <code>ihi = n</code> .
<code>job</code>	Must be ' <code>B</code> ', ' <code>S</code> ', ' <code>P</code> ', or ' <code>N</code> '. The default value is ' <code>B</code> '.

Application Notes

The errors are negligible, compared with those in subsequent computations.

If the matrix A is balanced by this routine, then any eigenvectors computed subsequently are eigenvectors of the matrix A'' and hence you must call `gebak` to transform them back to eigenvectors of A .

If the Schur vectors of A are required, do not call this routine with `job = 'S'` or '`B`', because then the balancing transformation is not orthogonal (not unitary for complex flavors).

If you call this routine with `job = 'P'`, then any Schur vectors computed subsequently are Schur vectors of the matrix A'' , and you need to call `gebak` (with `side = 'R'`) to transform them back to Schur vectors of A .

The total number of floating-point operations is proportional to n^2 .

?gebak

Transforms eigenvectors of a balanced matrix to those of the original nonsymmetric matrix.

Syntax

FORTRAN 77:

```
call sgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call dgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call cgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
call zgebak(job, side, n, ilo, ihi, scale, m, v, ldv, info)
```

Fortran 95:

```
call gebak(v, scale [,ilo] [,ihi] [,job] [,side] [,info])
```

C:

```
lapack_int LAPACKE_sgebak( int matrix_layout, char job, char side, lapack_int n,
                           lapack_int ilo, lapack_int ihi, const float* scale, lapack_int m, float* v, lapack_int ldv );
lapack_int LAPACKE_dgebak( int matrix_layout, char job, char side, lapack_int n,
                           lapack_int ilo, lapack_int ihi, const double* scale, lapack_int m, double* v,
                           lapack_int ldv );
lapack_int LAPACKE_cgebak( int matrix_layout, char job, char side, lapack_int n,
                           lapack_int ilo, lapack_int ihi, const float* scale, lapack_int m, lapack_complex_float* v,
                           lapack_int ldv );
lapack_int LAPACKE_zgebak( int matrix_layout, char job, char side, lapack_int n,
                           lapack_int ilo, lapack_int ihi, const double* scale, lapack_int m,
                           lapack_complex_double* v, lapack_int ldv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine is intended to be used after a matrix A has been balanced by a call to ?gebal, and eigenvectors of the balanced matrix A''_{22} have subsequently been computed. For a description of balancing, see [gebal](#). The balanced matrix A'' is obtained as $A'' = D^T P^* A * P^T \text{inv}(D)$, where P is a permutation matrix and D is a diagonal scaling matrix. This routine transforms the eigenvectors as follows:

If x is a right eigenvector of A'' , then $P^T \text{inv}(D) * x$ is a right eigenvector of A ; if x is a left eigenvector of A'' , then $P^T * D^* y$ is a left eigenvector of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>job</i>	CHARACTER*1. Must be 'N' or 'P' or 'S' or 'B'. The same parameter <i>job</i> as supplied to ?gebal.
<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', then left eigenvectors are transformed. If <i>side</i> = 'R', then right eigenvectors are transformed.
<i>n</i>	INTEGER. The number of rows of the matrix of eigenvectors ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. The values <i>ilo</i> and <i>ihi</i> , as returned by ?gebal. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, then <i>ilo</i> = 1 and <i>ihi</i> = 0.)
<i>scale</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors Array, DIMENSION at least max(1, <i>n</i>). Contains details of the permutations and/or the scaling factors used to balance the original general matrix, as returned by ?gebal.
<i>m</i>	INTEGER. The number of columns of the matrix of eigenvectors ($m \geq 0$).
<i>v</i>	REAL for sgebak DOUBLE PRECISION for dgebak COMPLEX for cgebak DOUBLE COMPLEX for zgebak. Arrays: <i>v</i> (<i>ldv, *</i>) contains the matrix of left or right eigenvectors to be transformed. The second dimension of <i>v</i> must be at least max(1, <i>m</i>).

ldv INTEGER. The leading dimension of *v*; at least $\max(1, n)$.

Output Parameters

v Overwritten by the transformed eigenvectors.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gebak` interface are the following:

v Holds the matrix *V* of size (*n,m*).

scale Holds the vector of length *n*.

ilo Default value for this argument is *ilo* = 1.

ih Default value for this argument is *ih* = *n*.

job Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.

side Must be 'L' or 'R'. The default value is 'L'.

Application Notes

The errors in this routine are negligible.

The approximate number of floating-point operations is approximately proportional to m^2n .

?hseqr

Computes all eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form.

Syntax

FORTRAN 77:

```
call shseqr(job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork, info)
call dhseqr(job, compz, n, ilo, ihi, h, ldh, wr, wi, z, ldz, work, lwork, info)
call chseqr(job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
call zhseqr(job, compz, n, ilo, ihi, h, ldh, w, z, ldz, work, lwork, info)
```

Fortran 95:

```
call hseqr(h, wr, wi [,ilo] [,ihi] [,z] [,job] [,compz] [,info])
call hseqr(h, w [,ilo] [,ihi] [,z] [,job] [,compz] [,info])
```

C:

```

lapack_int LAPACKE_shseqr( int matrix_layout, char job, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, float* h, lapack_int ldh, float* wr, float* wi, float*
z, lapack_int ldz );

lapack_int LAPACKE_dhseqr( int matrix_layout, char job, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, double* h, lapack_int ldh, double* wr, double* wi,
double* z, lapack_int ldz );

lapack_int LAPACKE_chseqr( int matrix_layout, char job, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, lapack_complex_float* h, lapack_int ldh,
lapack_complex_float* w, lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhseqr( int matrix_layout, char job, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, lapack_complex_double* h, lapack_int ldh,
lapack_complex_double* w, lapack_complex_double* z, lapack_int ldz );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally the Schur factorization, of an upper Hessenberg matrix H : $H = Z^* T^* Z^H$, where T is an upper triangular (or, for real flavors, quasi-triangular) matrix (the Schur form of H), and Z is the unitary or orthogonal matrix whose columns are the Schur vectors z_i .

You can also use this routine to compute the Schur factorization of a general matrix A which has been reduced to upper Hessenberg form H :

$$A = Q^* H^* Q^H, \text{ where } Q \text{ is unitary (orthogonal for real flavors);}$$

$$A = (QZ)^* T^* (QZ)^H.$$

In this case, after reducing A to Hessenberg form by [gehrd](#), call [orghr](#) to form Q explicitly and then pass Q to [?hseqr](#) with $compz = 'V'$.

You can also call [gebal](#) to balance the original matrix before reducing it to Hessenberg form by [?hseqr](#), so that the Hessenberg matrix H will have the structure:

$$\begin{bmatrix} H_{11} & H_{12} & H_{13} \\ 0 & H_{22} & H_{23} \\ 0 & 0 & H_{33} \end{bmatrix}$$

where H_{11} and H_{33} are upper triangular.

If so, only the central diagonal block H_{22} (in rows and columns ilo to ihi) needs to be further reduced to Schur form (the blocks H_{12} and H_{23} are also affected). Therefore the values of ilo and ihi can be supplied to [?hseqr](#) directly. Also, after calling this routine you must call [gebak](#) to permute the Schur vectors of the balanced matrix to those of the original matrix.

If [?gebal](#) has not been called, however, then ilo must be set to 1 and ihi to n . Note that if the Schur factorization of A is required, [?gebal](#) must not be called with $job = 'S'$ or $'B'$, because the balancing transformation is not unitary (for real flavors, it is not orthogonal).

?hseqr uses a multishift form of the upper Hessenberg QR algorithm. The Schur vectors are normalized so that $\|z_i\|_2 = 1$, but are determined only to within a complex factor of absolute value 1 (for the real flavors, to within a factor ± 1).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>job</i>	CHARACTER*1. Must be 'E' or 'S'. If <i>job</i> = 'E', then eigenvalues only are required. If <i>job</i> = 'S', then the Schur form <i>T</i> is required.
<i>compz</i>	CHARACTER*1. Must be 'N' or 'I' or 'V'. If <i>compz</i> = 'N', then no Schur vectors are computed (and the array <i>z</i> is not referenced). If <i>compz</i> = 'I', then the Schur vectors of <i>H</i> are computed (and the array <i>z</i> is initialized by the routine). If <i>compz</i> = 'V', then the Schur vectors of <i>A</i> are computed (and the array <i>z</i> must contain the matrix <i>Q</i> on entry).
<i>n</i>	INTEGER. The order of the matrix <i>H</i> ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. If <i>A</i> has been balanced by ?gebal, then <i>ilo</i> and <i>ihi</i> must contain the values returned by ?gebal. Otherwise, <i>ilo</i> must be set to 1 and <i>ihi</i> to <i>n</i> .
<i>h, z, work</i>	REAL for shseqr DOUBLE PRECISION for dhseqr COMPLEX for chseqr DOUBLE COMPLEX for zhseqr. Arrays: <i>h</i> (<i>ldh,*</i>) The <i>n</i> -by- <i>n</i> upper Hessenberg matrix <i>H</i> . The second dimension of <i>h</i> must be at least max(1, <i>n</i>). <i>z</i> (<i>ldz,*</i>) If <i>compz</i> = 'V', then <i>z</i> must contain the matrix <i>Q</i> from the reduction to Hessenberg form. If <i>compz</i> = 'I', then <i>z</i> need not be set. If <i>compz</i> = 'N', then <i>z</i> is not referenced. The second dimension of <i>z</i> must be at least max(1, <i>n</i>) if <i>compz</i> = 'V' or 'I'; at least 1 if <i>compz</i> = 'N'. <i>work</i> (<i>lwork</i>) is a workspace array. The dimension of <i>work</i> must be at least max (1, <i>n</i>). <i>ldh</i> INTEGER. The leading dimension of <i>h</i> ; at least max(1, <i>n</i>).

<i>ldz</i>	INTEGER. The leading dimension of <i>z</i> ; If <i>compz</i> = 'N', then <i>ldz</i> ≥ 1 . If <i>compz</i> = 'V' or 'I', then <i>ldz</i> $\geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . <i>lwork</i> $\geq \max(1, n)$ is sufficient and delivers very good and sometimes optimal performance. However, <i>lwork</i> as large as $11*n$ may be required for optimal performance. A workspace query is recommended to determine the optimal workspace size. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only estimates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for details.

Output Parameters

<i>w</i>	COMPLEX for <i>chseqr</i> DOUBLE COMPLEX for <i>zhseqr</i> . Array, DIMENSION at least $\max(1, n)$. Contains the computed eigenvalues, unless <i>info</i> >0. The eigenvalues are stored in the same order as on the diagonal of the Schur form <i>T</i> (if computed).
<i>wr, wi</i>	REAL for <i>shseqr</i> DOUBLE PRECISION for <i>dhseqr</i> Arrays, DIMENSION at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, unless <i>info</i> > 0. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first. The eigenvalues are stored in the same order as on the diagonal of the Schur form <i>T</i> (if computed).
<i>h</i>	If <i>info</i> = 0 and <i>job</i> = 'S', <i>h</i> contains the upper triangular matrix <i>T</i> from the Schur decomposition (the Schur form). If <i>info</i> = 0 and <i>job</i> = 'E', the contents of <i>h</i> are unspecified on exit. (The output value of <i>h</i> when <i>info</i> > 0 is given under the description of <i>info</i> below.)
<i>z</i>	If <i>compz</i> = 'V' and <i>info</i> = 0, then <i>z</i> contains <i>Q</i> * <i>Z</i> . If <i>compz</i> = 'I' and <i>info</i> = 0, then <i>z</i> contains the unitary or orthogonal matrix <i>Z</i> of the Schur vectors of <i>H</i> . If <i>compz</i> = 'N', then <i>z</i> is not referenced.
<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the optimal <i>lwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

If $info = i$, elements $1, 2, \dots, ilo-1$ and $i+1, i+2, \dots, n$ of wr and wi contain the real and imaginary parts of those eigenvalues that have been successively found.

If $info > 0$, and $job = 'E'$, then on exit, the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns ilo through $info$ of the final output value of H .

If $info > 0$, and $job = 'S'$, then on exit $(initial\ value\ of\ H)*U = U*(final\ value\ of\ H)$, where U is a unitary matrix. The final value of H is upper Hessenberg and triangular in rows and columns $info+1$ through ih .

If $info > 0$, and $compz = 'V'$, then on exit $(final\ value\ of\ Z) = (initial\ value\ of\ Z)*U$, where U is the unitary matrix (regardless of the value of job).

If $info > 0$, and $compz = 'I'$, then on exit $(final\ value\ of\ Z) = U$, where U is the unitary matrix (regardless of the value of job).

If $info > 0$, and $compz = 'N'$, then Z is not accessed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hseqr` interface are the following:

h	Holds the matrix H of size (n,n) .
wr	Holds the vector of length n . Used in real flavors only.
wi	Holds the vector of length n . Used in real flavors only.
w	Holds the vector of length n . Used in complex flavors only.
z	Holds the matrix Z of size (n,n) .
job	Must be ' E ' or ' S '. The default value is ' E '.
$compz$	If omitted, this argument is restored based on the presence of argument z as follows: $compz = 'I'$, if z is present, $compz = 'N'$, if z is omitted. If present, $compz$ must be equal to ' I ' or ' V ' and the argument z must also be present. Note that there will be an error condition if $compz$ is present and z omitted.

Application Notes

The computed Schur factorization is the exact factorization of a nearby matrix $H + E$, where $\|E\|_2 < O(\varepsilon) \|H\|_2 / s_i$, and ε is the machine precision.

If λ_i is an exact eigenvalue, and μ_i is the corresponding computed value, then $|\lambda_i - \mu_i| \leq c(n) * \varepsilon * \|H\|_2 / s_i$, where $c(n)$ is a modestly increasing function of n , and s_i is the reciprocal condition number of λ_i . The condition numbers s_i may be computed by calling `trsna`.

The total number of floating-point operations depends on how rapidly the algorithm converges; typical numbers are as follows.

If only eigenvalues are computed: $7n^3$ for real flavors

$25n^3$ for complex flavors.

If the Schur form is computed: $10n^3$ for real flavors
 $35n^3$ for complex flavors.

If the full Schur factorization is computed: $20n^3$ for real flavors
 $70n^3$ for complex flavors.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hsein

Computes selected eigenvectors of an upper Hessenberg matrix that correspond to specified eigenvalues.

Syntax

FORTRAN 77:

```
call shsein(side, eigsrc, initv, select, n, h, ldh, wr, wi, vl, ldvl, vr, ldvr, mm,
m, work, ifaill, ifailr, info)

call dhsein(side, eigsrc, initv, select, n, h, ldh, wr, wi, vl, ldvl, vr, ldvr, mm,
m, work, ifaill, ifailr, info)

call chsein(side, eigsrc, initv, select, n, h, ldh, w, vl, ldvl, vr, ldvr, mm, m,
work, rwork, ifaill, ifailr, info)

call zhsein(side, eigsrc, initv, select, n, h, ldh, w, vl, ldvl, vr, ldvr, mm, m,
work, rwork, ifaill, ifailr, info)
```

Fortran 95:

```
call hsein(h, wr, wi, select [, vl] [, vr] [,ifaill] [,ifailr] [,initv] [,eigsrc] [,m]
[,info])

call hsein(h, w, select [,vl] [,vr] [,ifaill] [,ifailr] [,initv] [,eigsrc] [,m] [,info])
```

C:

```
lapack_int LAPACKE_shsein( int matrix_layout, char side, char eigsrc, char initv,
lapack_logical* select, lapack_int n, const float* h, lapack_int ldh, float* wr,
const float* wi, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr, lapack_int
mm, lapack_int* m, lapack_int* ifaill, lapack_int* ifailr );

lapack_int LAPACKE_dhsein( int matrix_layout, char side, char eigsrc, char initv,
lapack_logical* select, lapack_int n, const double* h, lapack_int ldh, double* wr,
const double* wi, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr, lapack_int
mm, lapack_int* m, lapack_int* ifaill, lapack_int* ifailr );
```

```

lapack_int LAPACKE_chsein( int matrix_layout, char side, char eisrc, char initv,
const lapack_logical* select, lapack_int n, const lapack_complex_float* h, lapack_int
ldh, lapack_complex_float* w, lapack_complex_float* vl, lapack_int ldvl,
lapack_complex_float* vr, lapack_int ldvr, lapack_int mm, lapack_int* m, lapack_int*
ifaill, lapack_int* ifailr );

lapack_int LAPACKE_zhsein( int matrix_layout, char side, char eisrc, char initv,
const lapack_logical* select, lapack_int n, const lapack_complex_double* h, lapack_int
ldh, lapack_complex_double* w, lapack_complex_double* vl, lapack_int ldvl,
lapack_complex_double* vr, lapack_int ldvr, lapack_int mm, lapack_int* m, lapack_int*
ifaill, lapack_int* ifailr );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes left and/or right eigenvectors of an upper Hessenberg matrix H , corresponding to selected eigenvalues.

The right eigenvector x and the left eigenvector y , corresponding to an eigenvalue λ , are defined by: $H^*x = \lambda*x$ and $y^H*H = \lambda*y^H$ (or $H^H*y = \lambda^*y$). Here λ^* denotes the conjugate of λ .

The eigenvectors are computed by inverse iteration. They are scaled so that, for a real eigenvector x , $\max|x_i| = 1$, and for a complex eigenvector, $\max(|\text{Re}x_i| + |\text{Im}x_i|) = 1$.

If H has been formed by reduction of a general matrix A to upper Hessenberg form, then eigenvectors of H may be transformed to eigenvectors of A by [ormhr](#) or [unmhr](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be 'R' or 'L' or 'B'. If <i>side</i> = 'R', then only right eigenvectors are computed. If <i>side</i> = 'L', then only left eigenvectors are computed. If <i>side</i> = 'B', then all eigenvectors are computed.
<i>eisrc</i>	CHARACTER*1. Must be 'Q' or 'N'. If <i>eisrc</i> = 'Q', then the eigenvalues of H were found using hseqr ; thus if H has any zero sub-diagonal elements (and so is block triangular), then the j -th eigenvalue can be assumed to be an eigenvalue of the block containing the j -th row/column. This property allows the routine to perform inverse iteration on just one diagonal block. If <i>eisrc</i> = 'N', then no such assumption is made and the routine performs inverse iteration using the whole matrix.
<i>initv</i>	CHARACTER*1. Must be 'N' or 'U'. If <i>initv</i> = 'N', then no initial estimates for the selected eigenvectors are supplied.

If *initv* = 'U', then initial estimates for the selected eigenvectors are supplied in *vl* and/or *vr*.

select

LOGICAL.

Array, DIMENSION at least max (1, *n*). Specifies which eigenvectors are to be computed.

For real flavors:

To obtain the real eigenvector corresponding to the real eigenvalue *wr(j)*, set *select(j)* to .TRUE..

To select the complex eigenvector corresponding to the complex eigenvalue (*wr(j)*, *wi(j)*) with complex conjugate (*wr(j+1)*, *wi(j+1)*), set *select(j)* and/or *select(j+1)* to .TRUE.; the eigenvector corresponding to the first eigenvalue in the pair is computed.

For complex flavors:

To select the eigenvector corresponding to the eigenvalue *w(j)*, set *select(j)* to .TRUE..

n

INTEGER. The order of the matrix *H* (*n* \geq 0).

h, vl, vr,

REAL for shsein

DOUBLE PRECISION for dhsein

COMPLEX for chsein

DOUBLE COMPLEX for zhsein.

Arrays:

h(Idh,)* The *n*-by-*n* upper Hessenberg matrix *H*. If an *NAN* value is detected in *h*, the routine returns with *info* = -6.

The second dimension of *h* must be at least max(1, *n*).

vl(Idvl,)*

If *initv* = 'V' and *side* = 'L' or 'B', then *vl* must contain starting vectors for inverse iteration for the left eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.

If *initv* = 'N', then *vl* need not be set.

The second dimension of *vl* must be at least max(1, *mm*) if *side* = 'L' or 'B' and at least 1 if *side* = 'R'.

The array *vl* is not referenced if *side* = 'R'.

vr(Idvr,)*

If *initv* = 'V' and *side* = 'R' or 'B', then *vr* must contain starting vectors for inverse iteration for the right eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.

If *initv* = 'N', then *vr* need not be set.

The second dimension of *vr* must be at least max(1, *mm*) if *side* = 'R' or 'B' and at least 1 if *side* = 'L'.

The array *vr* is not referenced if *side* = 'L'.
work(*) is a workspace array.
 DIMENSION at least max (1, *n**(*n*+2)) for real flavors and at least max (1, *n***n*) for complex flavors.

ldh
 INTEGER. The leading dimension of *h*; at least max(1, *n*).

w
 COMPLEX for chsein
 DOUBLE COMPLEX for zhsein.
 Array, DIMENSION at least max (1, *n*).
 Contains the eigenvalues of the matrix *H*.
 If *eigsrc* = 'Q', the array must be exactly as returned by ?hseqr.

wr, wi
 REAL for shsein
 DOUBLE PRECISION for dhsein
 Arrays, DIMENSION at least max (1, *n*) each.
 Contain the real and imaginary parts, respectively, of the eigenvalues of the matrix *H*. Complex conjugate pairs of values must be stored in consecutive elements of the arrays. If *eigsrc* = 'Q', the arrays must be exactly as returned by ?hseqr.

ldvl
 INTEGER. The leading dimension of *vl*.
 If *side* = 'L' or 'B', *ldvl* $\geq \max(1, n)$.
 If *side* = 'R', *ldvl* ≥ 1 .

ldvr
 INTEGER. The leading dimension of *vr*.
 If *side* = 'R' or 'B', *ldvr* $\geq \max(1, n)$.
 If *side* = 'L', *ldvr* ≥ 1 .

mm
 INTEGER. The number of columns in *vl* and/or *vr*.
 Must be at least *m*, the actual number of columns required (see *Output Parameters* below).
 For real flavors, *m* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector (see *select*).
 For complex flavors, *m* is the number of selected eigenvectors (see *select*).
 Constraint:
 $0 \leq mm \leq n$.

rwork
 REAL for chsein
 DOUBLE PRECISION for zhsein.
 Array, DIMENSION at least max (1, *n*).

Output Parameters

select Overwritten for real flavors only.

If a complex eigenvector was selected as specified above, then `select(j)` is set to .TRUE. and `select(j+1)` to .FALSE.

w

The real parts of some elements of *w* may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

wr

Some elements of *wr* may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

vl, vr

If `side = 'L'` or `'B'`, *vl* contains the computed left eigenvectors (as specified by `select`).

If `side = 'R'` or `'B'`, *vr* contains the computed right eigenvectors (as specified by `select`).

The eigenvectors are stored consecutively in the columns of the array, in the same order as their eigenvalues.

For real flavors: a real eigenvector corresponding to a selected real eigenvalue occupies one column; a complex eigenvector corresponding to a selected complex eigenvalue occupies two columns: the first column holds the real part and the second column holds the imaginary part.

m

INTEGER. *For real flavors:* the number of columns of *vl* and/or *vr* required to store the selected eigenvectors.

For complex flavors: the number of selected eigenvectors.

ifaill, ifailr

INTEGER.

Arrays, DIMENSION at least `max(1, mm)` each.

ifaill(i) = 0 if the *i*th column of *vl* converged;

ifaill(i) = j > 0 if the eigenvector stored in the *i*-th column of *vl* (corresponding to the *j*th eigenvalue) failed to converge.

ifailr(i) = 0 if the *i*th column of *vr* converged;

ifailr(i) = j > 0 if the eigenvector stored in the *i*-th column of *vr* (corresponding to the *j*th eigenvalue) failed to converge.

For real flavors: if the *i*th and *(i+1)*th columns of *vl* contain a selected complex eigenvector, then *ifaill(i)* and *ifaill(i+1)* are set to the same value. A similar rule holds for *vr* and *ifailr*.

The array *ifaill* is not referenced if `side = 'R'`. The array *ifailr* is not referenced if `side = 'L'`.

info

INTEGER.

If *info = 0*, the execution is successful.

If *info = -i*, the *i*-th parameter had an illegal value.

If *info > 0*, then *i* eigenvectors (as indicated by the parameters *ifaill* and/or *ifailr* above) failed to converge. The corresponding columns of *vl* and/or *vr* contain no useful information.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hsein` interface are the following:

<code>h</code>	Holds the matrix H of size (n,n) .
<code>wr</code>	Holds the vector of length n . Used in real flavors only.
<code>wi</code>	Holds the vector of length n . Used in real flavors only.
<code>w</code>	Holds the vector of length n . Used in complex flavors only.
<code>select</code>	Holds the vector of length n .
<code>vl</code>	Holds the matrix VL of size (n,mm) .
<code>vr</code>	Holds the matrix VR of size (n,mm) .
<code>ifail1</code>	Holds the vector of length (mm) . Note that there will be an error condition if <code>ifail1</code> is present and <code>vl</code> is omitted.
<code>ifailr</code>	Holds the vector of length (mm) . Note that there will be an error condition if <code>ifailr</code> is present and <code>vr</code> is omitted.
<code>initv</code>	Must be ' <code>N</code> ' or ' <code>U</code> '. The default value is ' <code>N</code> '.
<code>eigsrc</code>	Must be ' <code>N</code> ' or ' <code>Q</code> '. The default value is ' <code>N</code> '.
<code>side</code>	Restored based on the presence of arguments <code>vl</code> and <code>vr</code> as follows: <code>side = 'B'</code> , if both <code>vl</code> and <code>vr</code> are present, <code>side = 'L'</code> , if <code>vl</code> is present and <code>vr</code> omitted, <code>side = 'R'</code> , if <code>vl</code> is omitted and <code>vr</code> present, Note that there will be an error condition if both <code>vl</code> and <code>vr</code> are omitted.

Application Notes

Each computed right eigenvector x_i is the exact eigenvector of a nearby matrix $A + E_i$, such that $\|E_i\| < O(\epsilon) \|A\|$. Hence the residual is small:

$$\|Ax_i - \lambda_i x_i\| = O(\epsilon) \|A\|.$$

However, eigenvectors corresponding to close or coincident eigenvalues may not accurately span the relevant subspaces.

Similar remarks apply to computed left eigenvectors.

?trevc

Computes selected eigenvectors of an upper (quasi-)triangular matrix computed by `?hseqr`.

Syntax

FORTRAN 77:

```
call strevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, info)
```

```
call dtrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, info)
call ctrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, rwork,
info)
call ztrevc(side, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, mm, m, work, rwork,
info)
```

Fortran 95:

```
call trevc(t [, howmny] [,select] [,vl] [,vr] [,m] [,info])
```

C:

```
lapack_int LAPACKE_strevc( int matrix_layout, char side, char howmny, lapack_logical*
select, lapack_int n, const float* t, lapack_int ldt, float* vl, lapack_int ldvl,
float* vr, lapack_int ldvr, lapack_int mm, lapack_int* m );
lapack_int LAPACKE_dtrevc( int matrix_layout, char side, char howmny, lapack_logical*
select, lapack_int n, const double* t, lapack_int ldt, double* vl, lapack_int ldvl,
double* vr, lapack_int ldvr, lapack_int mm, lapack_int* m );
lapack_int LAPACKE_ctrevc( int matrix_layout, char side, char howmny,
const lapack_logical* select, lapack_int n, lapack_complex_float* t, lapack_int ldt,
lapack_complex_float* vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr,
lapack_int mm, lapack_int* m );
lapack_int LAPACKE_ztrevc( int matrix_layout, char side, char howmny,
const lapack_logical* select, lapack_int n, lapack_complex_double* t, lapack_int ldt,
lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int ldvr,
lapack_int mm, lapack_int* m );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes some or all of the right and/or left eigenvectors of an upper triangular matrix T (or, for real flavors, an upper quasi-triangular matrix T). Matrices of this type are produced by the Schur factorization of a general matrix: $A = Q^* T^* Q^H$, as computed by [hseqr](#).

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w , are defined by:

$T^*x = w^*x$, $y^H T = w^*y^H$, where y^H denotes the conjugate transpose of y .

The eigenvalues are not input to this routine, but are read directly from the diagonal blocks of T .

This routine returns the matrices X and/or Y of right and left eigenvectors of T , or the products Q^*X and/or Q^*Y , where Q is an input matrix.

If Q is the orthogonal/unitary factor that reduces a matrix A to Schur form T , then Q^*X and Q^*Y are the matrices of right and left eigenvectors of A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

side	CHARACTER*1. Must be 'R' or 'L' or 'B'.
------	---

If *side* = 'R', then only right eigenvectors are computed.

If *side* = 'L', then only left eigenvectors are computed.

If *side* = 'B', then all eigenvectors are computed.

howmny

CHARACTER*1. Must be 'A' or 'B' or 'S'.

If *howmny* = 'A', then all eigenvectors (as specified by *side*) are computed.

If *howmny* = 'B', then all eigenvectors (as specified by *side*) are computed and backtransformed by the matrices supplied in *vl* and *vr*.

If *howmny* = 'S', then selected eigenvectors (as specified by *side* and *select*) are computed.

select

LOGICAL.

Array, DIMENSION at least max (1, *n*).

If *howmny* = 'S', *select* specifies which eigenvectors are to be computed.

If *howmny* = 'A' or 'B', *select* is not referenced.

For real flavors:

If *omega(j)* is a real eigenvalue, the corresponding real eigenvector is computed if *select(j)* is .TRUE..

If *omega(j)* and *omega(j+1)* are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either *select(j)* or *select(j+1)* is .TRUE., and on exit *select(j)* is set to .TRUE.and *select(j+1)* is set to .FALSE..

For complex flavors:

The eigenvector corresponding to the *j*-th eigenvalue is computed if *select(j)* is .TRUE..

n

INTEGER. The order of the matrix *T* (*n* \geq 0).

t, vl, vr

REAL for strevc

DOUBLE PRECISION for dtrevc

COMPLEX for ctrevc

DOUBLE COMPLEX for ztrevc.

Arrays:

*t(l, l, *)* contains the *n*-by-*n* matrix *T* in Schur canonical form. For complex flavors *ctrevc* and *ztrevc*, contains the upper triangular matrix *T*.

The second dimension of *t* must be at least max(1, *n*).

*vl(l, l, *)*

If *howmny* = 'B' and *side* = 'L' or 'B', then *vl* must contain an *n*-by-*n* matrix *Q* (usually the matrix of Schur vectors returned by ?hseqr).

If *howmny* = 'A' or 'S', then *vl* need not be set.

The second dimension of *vl* must be at least max(1, *mm*) if *side* = 'L' or 'B' and at least 1 if *side* = 'R'.

The array *vl* is not referenced if *side* = 'R'.

vr (*ldvr,**)

If *howmny* = 'B' and *side* = 'R' or 'B', then *vr* must contain an *n*-by-*n* matrix *Q* (usually the matrix of Schur vectors returned by ?hseqr). .

If *howmny* = 'A' or 'S', then *vr* need not be set.

The second dimension of *vr* must be at least $\max(1, mm)$ if *side* = 'R' or 'B' and at least 1 if *side* = 'L'.

The array *vr* is not referenced if *side* = 'L'.

work()* is a workspace array.

DIMENSION at least $\max(1, 3*n)$ for real flavors and at least $\max(1, 2*n)$ for complex flavors.

ldt

INTEGER. The leading dimension of *t*; at least $\max(1, n)$.

ldvl

INTEGER. The leading dimension of *vl*.

If *side* = 'L' or 'B', *ldvl* $\geq n$.

If *side* = 'R', *ldvl* ≥ 1 .

ldvr

INTEGER. The leading dimension of *vr*.

If *side* = 'R' or 'B', *ldvr* $\geq n$.

If *side* = 'L', *ldvr* ≥ 1 .

mm

INTEGER. The number of columns in the arrays *vl* and/or *vr*. Must be at least *m* (the precise number of columns required).

If *howmny* = 'A' or 'B', *m* = *n*.

If *howmny* = 'S': for real flavors, *m* is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector; for complex flavors, *m* is the number of selected eigenvectors (see *select*).

Constraint: $0 \leq m \leq n$.

rwork

REAL for *ctrevc*

DOUBLE PRECISION for *ztrrevc*.

Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

select

If a complex eigenvector of a real matrix was selected as specified above, then *select(j)* is set to .TRUE. and *select(j+1)* to .FALSE..

t

COMPLEX for *ctrevc*

DOUBLE COMPLEX for *ztrrevc*.

ctrevc/ztrrevc modify the *t*(*ldt,**) array, which is restored on exit.

vl, vr

If *side* = 'L' or 'B', *vl* contains the computed left eigenvectors (as specified by *howmny* and *select*).

If $\text{side} = \text{'R'}$ or 'B' , vr contains the computed right eigenvectors (as specified by howmny and select).

The eigenvectors are stored consecutively in the columns of the array, in the same order as their eigenvalues.

For real flavors: corresponding to each real eigenvalue is a real eigenvector, occupying one column; corresponding to each complex conjugate pair of eigenvalues is a complex eigenvector, occupying two columns; the first column holds the real part and the second column holds the imaginary part.

m

INTEGER.

For complex flavors: the number of selected eigenvectors.

If $\text{howmny} = \text{'A'}$ or 'B' , m is set to n .

For real flavors: the number of columns of vl and/or vr actually used to store the selected eigenvectors.

If $\text{howmny} = \text{'A'}$ or 'B' , m is set to n .

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trevc` interface are the following:

t Holds the matrix T of size (n,n) .

$select$ Holds the vector of length n .

vl Holds the matrix VL of size (n,mm) .

vr Holds the matrix VR of size (n,mm) .

$side$ If omitted, this argument is restored based on the presence of arguments vl and vr as follows:

$side = \text{'B'}$, if both vl and vr are present,

$side = \text{'L'}$, if vr is omitted,

$side = \text{'R'}$, if vl is omitted.

Note that there will be an error condition if both vl and vr are omitted.

$howmny$ If omitted, this argument is restored based on the presence of argument $select$ as follows:

$howmny = \text{'V'}$, if q is present,

$howmny = \text{'N'}$, if q is omitted.

If present, $\text{vect} = \text{'V'}$ or 'U' and the argument q must also be present.

Note that there will be an error condition if both $select$ and $howmny$ are present.

Application Notes

If x_i is an exact right eigenvector and y_i is the corresponding computed eigenvector, then the angle $\theta(y_i, x_i)$ between them is bounded as follows: $\theta(y_i, x_i) \leq (c(n)\epsilon \|T\|_2)/\text{sep}_i$ where sep_i is the reciprocal condition number of x_i . The condition number sep_i may be computed by calling ?trsna.

?trsna

Estimates condition numbers for specified eigenvalues and right eigenvectors of an upper (quasi-) triangular matrix.

Syntax

FORTRAN 77:

```
call strsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work,
ldwork, iwork, info)

call dtrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work,
ldwork, iwork, info)

call ctrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work,
ldwork, rwork, info)

call ztrsna(job, howmny, select, n, t, ldt, vl, ldvl, vr, ldvr, s, sep, mm, m, work,
ldwork, rwork, info)
```

Fortran 95:

```
call trsna(t [, s] [,sep] [,vl] [,vr] [,select] [,m] [,info])
```

C:

```
lapack_int LAPACKE_strsna( int matrix_layout, char job, char howmny,
const lapack_logical* select, lapack_int n, const float* t, lapack_int ldt,
const float* vl, lapack_int ldvl, const float* vr, lapack_int ldvr, float* s, float*
sep, lapack_int mm, lapack_int* m );

lapack_int LAPACKE_dtrsna( int matrix_layout, char job, char howmny,
const lapack_logical* select, lapack_int n, const double* t, lapack_int ldt,
const double* vl, lapack_int ldvl, const double* vr, lapack_int ldvr, double* s,
double* sep, lapack_int mm, lapack_int* m );

lapack_int LAPACKE_ctrsna( int matrix_layout, char job, char howmny,
const lapack_logical* select, lapack_int n, const lapack_complex_float* t, lapack_int
ldt, const lapack_complex_float* vl, lapack_int ldvl, const lapack_complex_float* vr,
lapack_int ldvr, float* s, float* sep, lapack_int mm, lapack_int* m );

lapack_int LAPACKE_ztrsna( int matrix_layout, char job, char howmny,
const lapack_logical* select, lapack_int n, const lapack_complex_double* t, lapack_int
ldt, const lapack_complex_double* vl, lapack_int ldvl, const lapack_complex_double* vr,
lapack_int ldvr, double* s, double* sep, lapack_int mm, lapack_int* m );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine estimates condition numbers for specified eigenvalues and/or right eigenvectors of an upper triangular matrix T (or, for real flavors, upper quasi-triangular matrix T in canonical Schur form). These are the same as the condition numbers of the eigenvalues and right eigenvectors of an original matrix $A = Z^*T^*Z^H$ (with unitary or, for real flavors, orthogonal Z), from which T may have been derived.

The routine computes the reciprocal of the condition number of an eigenvalue $\lambda(i)$ as $s_i = \|v^T u\| / (\|u\|_E \|v\|_E)$ for real flavors and $s_i = \|v^H u\| / (\|u\|_E \|v\|_E)$ for complex flavors,

where:

- u and v are the right and left eigenvectors of T , respectively, corresponding to `lambda(i)`.
 - v^T/v^H denote transpose/conjugate transpose of v , respectively.

This reciprocal condition number always lies between zero (ill-conditioned) and one (well-conditioned).

An approximate error estimate for a computed eigenvalue λ_i is then given by $\varepsilon^* \|T\| / s_i$, where ε is the *machine precision*.

To estimate the reciprocal of the condition number of the right eigenvector corresponding to `lambda(i)`, the routine first calls `trexc` to reorder the eigenvalues so that `lambda(i)` is in the leading position:

10.000-10.000

The reciprocal condition number of the eigenvector is then estimated as sep_i , the smallest singular value of the matrix $T_{22} - \lambda(i) * I$. This number ranges from zero (ill-conditioned) to very large (well-conditioned).

An approximate error estimate for a computed right eigenvector u corresponding to $\lambda(i)$ is then given by $\varepsilon^* \|T\| / \text{sep}_i$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

job CHARACTER*1. Must be 'E' or 'V' or 'B'.
 If *job* = 'E', then condition numbers for eigenvalues only are computed.
 If *job* = 'V', then condition numbers for eigenvectors only are computed.
 If *job* = 'B', then condition numbers for both eigenvalues and eigenvectors are computed.

howmny CHARACTER*1. Must be 'A' or 'S'.
 If *howmny* = 'A', then the condition numbers for all eigenpairs are computed.
 If *howmny* = 'S', then condition numbers for selected eigenpairs (as specified by *select*) are computed.

select LOGICAL.

Array, DIMENSION at least $\max(1, n)$ if $howmany = 'S'$ and at least 1 otherwise.

Specifies the eigenpairs for which condition numbers are to be computed if *howmny* = 'S'.

For real flavors:

To select condition numbers for the eigenpair corresponding to the real eigenvalue *lambda(j)*, *select(j)* must be set .TRUE.;

to select condition numbers for the eigenpair corresponding to a complex conjugate pair of eigenvalues *lambda(j)* and *lambda(j+1)*, *select(j)* and/or *select(j+1)* must be set .TRUE. .

For complex flavors

To select condition numbers for the eigenpair corresponding to the eigenvalue *lambda(j)*, *select(j)* must be set .TRUE. *select* is not referenced if *howmny* = 'A'.

n INTEGER. The order of the matrix *T* (*n* \geq 0).

t, vl, vr, work REAL for strsna

DOUBLE PRECISION for dtrsna

COMPLEX for ctrsna

DOUBLE COMPLEX for ztrsna.

Arrays:

t(ldt,)* contains the *n*-by-*n* matrix *T*.

The second dimension of *t* must be at least max(1, *n*).

vl(ldvl,)*

If *job* = 'E' or 'B', then *vl* must contain the left eigenvectors of *T* (or of any matrix *Q***T***Q*^H with *Q* unitary or orthogonal) corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vl*, as returned by *trevc* or *hsein*.

The second dimension of *vl* must be at least max(1, *mm*) if *job* = 'E' or 'B' and at least 1 if *job* = 'V'.

The array *vl* is not referenced if *job* = 'V'.

vr(ldvr,)*

If *job* = 'E' or 'B', then *vr* must contain the right eigenvectors of *T* (or of any matrix *Q***T***Q*^H with *Q* unitary or orthogonal) corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vr*, as returned by *trevc* or *hsein*.

The second dimension of *vr* must be at least max(1, *mm*) if *job* = 'E' or 'B' and at least 1 if *job* = 'V'.

The array *vr* is not referenced if *job* = 'V'.

work is a workspace array, its dimension (*ldwork*, *n*+6).

The array *work* is not referenced if *job* = 'E'.

ldt INTEGER. The leading dimension of *t*; at least max(1, *n*).

ldvl INTEGER. The leading dimension of *vl*.

If $job = 'E'$ or $'B'$, $ldvl \geq \max(1, n)$.
If $job = 'V'$, $ldvl \geq 1$.

$ldvr$ INTEGER. The leading dimension of vr .
If $job = 'E'$ or $'B'$, $ldvr \geq \max(1, n)$.
If $job = 'R'$, $ldvr \geq 1$.

mm INTEGER. The number of elements in the arrays s and sep , and the number of columns in vl and vr (if used). Must be at least m (the precise number required).
If $howmny = 'A'$, $m = n$;
if $howmny = 'S'$, for real flavors m is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues.
for complex flavors m is the number of selected eigenpairs (see *select*).
Constraint:
 $0 \leq m \leq n$.

$ldwork$ INTEGER. The leading dimension of $work$.
If $job = 'V'$ or $'B'$, $ldwork \geq \max(1, n)$.
If $job = 'E'$, $ldwork \geq 1$.

$rwork$ REAL for ctrnsna , ztrnsna .
Array, DIMENSION at least $\max(1, n)$. The array is not referenced if $job = 'E'$.

$iwork$ INTEGER for strnsna , dtrnsna .
Array, DIMENSION at least $\max(1, 2*(n - 1))$. The array is not referenced if $job = 'E'$.

Output Parameters

s REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.
Array, DIMENSION at least $\max(1, mm)$ if $job = 'E'$ or $'B'$ and at least 1 if $job = 'V'$.
Contains the reciprocal condition numbers of the selected eigenvalues if $job = 'E'$ or $'B'$, stored in consecutive elements of the array. Thus $s(j)$, $sep(j)$ and the j -th columns of vl and vr all correspond to the same eigenpair (but not in general the j th eigenpair unless all eigenpairs have been selected).
For real flavors: for a complex conjugate pair of eigenvalues, two consecutive elements of S are set to the same value. The array s is not referenced if $job = 'V'$.

sep REAL for single-precision flavors
DOUBLE PRECISION for double-precision flavors.

Array, DIMENSION at least $\max(1, mm)$ if $job = 'V'$ or ' B ' and at least 1 if $job = 'E'$. Contains the estimated reciprocal condition numbers of the selected right eigenvectors if $job = 'V'$ or ' B ', stored in consecutive elements of the array.

For real flavors: for a complex eigenvector, two consecutive elements of sep are set to the same value; if the eigenvalues cannot be reordered to compute $sep(j)$, then $sep(j)$ is set to zero; this can only occur when the true value would be very small anyway. The array sep is not referenced if $job = 'E'$.

m

INTEGER.

For complex flavors: the number of selected eigenpairs.If $howmny = 'A'$, m is set to n .*For real flavors:* the number of elements of s and/or sep actually used to store the estimated condition numbers.If $howmny = 'A'$, m is set to n .*info*

INTEGER.

If $info = 0$, the execution is successful.If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trsna` interface are the following:

<i>t</i>	Holds the matrix T of size (n,n) .
<i>s</i>	Holds the vector of length (mm) .
<i>sep</i>	Holds the vector of length (mm) .
<i>vl</i>	Holds the matrix VL of size (n,mm) .
<i>vr</i>	Holds the matrix VR of size (n,mm) .
<i>select</i>	Holds the vector of length n .
<i>job</i>	Restored based on the presence of arguments <i>s</i> and <i>sep</i> as follows: $job = 'B'$, if both <i>s</i> and <i>sep</i> are present, $job = 'E'$, if <i>s</i> is present and <i>sep</i> omitted, $job = 'V'$, if <i>s</i> is omitted and <i>sep</i> present. Note an error condition if both <i>s</i> and <i>sep</i> are omitted.
<i>howmny</i>	Restored based on the presence of the argument <i>select</i> as follows: $howmny = 'S'$, if <i>select</i> is present, $howmny = 'A'$, if <i>select</i> is omitted.

Note that the arguments s , vl , and vr must either be all present or all omitted.

Otherwise, an error condition is observed.

Application Notes

The computed values sep_i may overestimate the true value, but seldom by a factor of more than 3.

?trexc

Reorders the Schur factorization of a general matrix.

Syntax

FORTRAN 77:

```
call strexc(compq, n, t, ldt, q, ldq, ifst, ilst, work, info)
call dtrexc(compq, n, t, ldt, q, ldq, ifst, ilst, work, info)
call ctrexc(compq, n, t, ldt, q, ldq, ifst, ilst, info)
call ztrexc(compq, n, t, ldt, q, ldq, ifst, ilst, info)
```

Fortran 95:

```
call trexc(t, ifst, ilst [,q] [,info])
```

C:

```
lapack_int LAPACKE_strexc( int matrix_layout, char compq, lapack_int n, float* t,
                           lapack_int ldt, float* q, lapack_int ldq, lapack_int* ifst, lapack_int* ilst );
lapack_int LAPACKE_dtrexc( int matrix_layout, char compq, lapack_int n, double* t,
                           lapack_int ldt, double* q, lapack_int ldq, lapack_int* ifst, lapack_int* ilst );
lapack_int LAPACKE_ctrexc( int matrix_layout, char compq, lapack_int n,
                           lapack_complex_float* t, lapack_int ldt, lapack_complex_float* q, lapack_int ldq,
                           lapack_int ifst, lapack_int ilst );
lapack_int LAPACKE_ztrexc( int matrix_layout, char compq, lapack_int n,
                           lapack_complex_double* t, lapack_int ldt, lapack_complex_double* q, lapack_int ldq,
                           lapack_int ifst, lapack_int ilst );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reorders the Schur factorization of a general matrix $A = Q^H T Q^H$, so that the diagonal element or block of T with row index $ifst$ is moved to row $ilst$.

The reordered Schur form S is computed by an unitary (or, for real flavors, orthogonal) similarity transformation: $S = Z^H T Z$. Optionally the updated matrix P of Schur vectors is computed as $P = Q Z$, giving $A = P S P^H$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>compq</i>	CHARACTER*1. Must be 'V' or 'N'. If <i>compq</i> = 'V', then the Schur vectors (<i>Q</i>) are updated. If <i>compq</i> = 'N', then no Schur vectors are updated.
<i>n</i>	INTEGER. The order of the matrix <i>T</i> (<i>n</i> ≥ 0).
<i>t, q</i>	REAL for <i>strexc</i> DOUBLE PRECISION for <i>dtrexc</i> COMPLEX for <i>ctrexc</i> DOUBLE COMPLEX for <i>ztrexc</i> . Arrays: <i>t</i> (<i>ldt</i> ,*) contains the <i>n</i> -by- <i>n</i> matrix <i>T</i> . The second dimension of <i>t</i> must be at least $\max(1, n)$. <i>q</i> (<i>ldq</i> ,*) If <i>compq</i> = 'V', then <i>q</i> must contain <i>Q</i> (Schur vectors). If <i>compq</i> = 'N', then <i>q</i> is not referenced. The second dimension of <i>q</i> must be at least $\max(1, n)$ if <i>compq</i> = 'V' and at least 1 if <i>compq</i> = 'N'.
<i>ldt</i>	INTEGER. The leading dimension of <i>t</i> ; at least $\max(1, n)$.
<i>ldq</i>	INTEGER. The leading dimension of <i>q</i> ; If <i>compq</i> = 'N', then <i>ldq</i> ≥ 1. If <i>compq</i> = 'V', then <i>ldq</i> ≥ $\max(1, n)$.
<i>ifst, ilst</i>	INTEGER. $1 \leq \text{ifst} \leq n$; $1 \leq \text{ilst} \leq n$. Must specify the reordering of the diagonal elements (or blocks, which is possible for real flavors) of the matrix <i>T</i> . The element (or block) with row index <i>ifst</i> is moved to row <i>ilst</i> by a sequence of exchanges between adjacent elements (or blocks).
<i>work</i>	REAL for <i>strexc</i> DOUBLE PRECISION for <i>dtrexc</i> . Array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>t</i>	Overwritten by the updated matrix <i>S</i> .
<i>q</i>	If <i>compq</i> = 'V', <i>q</i> contains the updated matrix of Schur vectors.
<i>ifst, ilst</i>	Overwritten for real flavors only. If <i>ifst</i> pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; <i>ilst</i> always points to the first row of the block in its final position (which may differ from its input value by ±1).

<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.
-------------	--

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `trexc` interface are the following:

<i>t</i>	Holds the matrix T of size (n,n).
<i>q</i>	Holds the matrix Q of size (n,n).
<i>compq</i>	Restored based on the presence of the argument <i>q</i> as follows: <i>compq</i> = 'V', if <i>q</i> is present, <i>compq</i> = 'N', if <i>q</i> is omitted.

Application Notes

The computed matrix S is exactly similar to a matrix $T+E$, where $\|E\|_2 = O(\epsilon) * \|T\|_2$, and ϵ is the machine precision.

Note that if a 2 by 2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2 by 2 block to break into two 1 by 1 blocks, that is, for a pair of complex eigenvalues to become purely real.

The values of eigenvalues however are never changed by the re-ordering.

The approximate number of floating-point operations is

for real flavors:	$6n(ifst-ilst)$ if <i>compq</i> = 'N'; $12n(ifst-ilst)$ if <i>compq</i> = 'V';
for complex flavors:	$20n(ifst-ilst)$ if <i>compq</i> = 'N'; $40n(ifst-ilst)$ if <i>compq</i> = 'V'.

?trsen

Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.

Syntax

FORTRAN 77:

```
call strsen(job, compq, select, n, t, ldt, q, ldq, wr, wi, m, s, sep, work, lwork,
iwork, liwork, info)

call dtrsen(job, compq, select, n, t, ldt, q, ldq, wr, wi, m, s, sep, work, lwork,
iwork, liwork, info)

call ctrsen(job, compq, select, n, t, ldt, q, ldq, w, m, s, sep, work, lwork, info)
```

```
call ztrsen(job, compq, select, n, t, ldt, q, ldq, w, m, s, sep, work, lwork, info)
```

Fortran 95:

```
call trsen(t, select [,wr] [,wi] [,m] [,s] [,sep] [,q] [,info])
call trsen(t, select [,w] [,m] [,s] [,sep] [,q] [,info])
```

C:

```
lapack_int LAPACKE_ztrsen( int matrix_layout, char job, char compq,
const lapack_logical* select, lapack_int n, float* t, lapack_int ldt, float* q,
lapack_int ldq, float* wr, float* wi, lapack_int* m, float* s, float* sep );
lapack_int LAPACKE_dtrsen( int matrix_layout, char job, char compq,
const lapack_logical* select, lapack_int n, double* t, lapack_int ldt, double* q,
lapack_int ldq, double* wr, double* wi, lapack_int* m, double* s, double* sep );
lapack_int LAPACKE_ctrsen( int matrix_layout, char job, char compq,
const lapack_logical* select, lapack_int n, lapack_complex_float* t, lapack_int ldt,
lapack_complex_float* q, lapack_int ldq, lapack_complex_float* w, lapack_int* m, float*
s, float* sep );
lapack_int LAPACKE_ztrsen( int matrix_layout, char job, char compq,
const lapack_logical* select, lapack_int n, lapack_complex_double* t, lapack_int ldt,
lapack_complex_double* q, lapack_int ldq, lapack_complex_double* w, lapack_int* m,
double* s, double* sep );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reorders the Schur factorization of a general matrix $A = Q*T*Q^T$ (for real flavors) or $A = Q*T*Q^H$ (for complex flavors) so that a selected cluster of eigenvalues appears in the leading diagonal elements (or, for real flavors, diagonal blocks) of the Schur form. The reordered Schur form R is computed by a unitary (orthogonal) similarity transformation: $R = Z^H*T*Z$. Optionally the updated matrix P of Schur vectors is computed as $P = Q*Z$, giving $A = P*R*P^H$.

Let

... :::::::

where the selected eigenvalues are precisely the eigenvalues of the leading m -by- m submatrix T_{11} . Let P be correspondingly partitioned as $(Q_1 \ Q_2)$ where Q_1 consists of the first m columns of Q . Then $A*Q_1 = Q_1*T_{11}$, and so the m columns of Q_1 form an orthonormal basis for the invariant subspace corresponding to the selected cluster of eigenvalues.

Optionally the routine also computes estimates of the reciprocal condition numbers of the average of the cluster of eigenvalues and of the invariant subspace.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>job</i>	CHARACTER*1. Must be 'N' or 'E' or 'V' or 'B'. If <i>job</i> = 'N', then no condition numbers are required. If <i>job</i> = 'E', then only the condition number for the cluster of eigenvalues is computed. If <i>job</i> = 'V', then only the condition number for the invariant subspace is computed. If <i>job</i> = 'B', then condition numbers for both the cluster and the invariant subspace are computed.
<i>compq</i>	CHARACTER*1. Must be 'V' or 'N'. If <i>compq</i> = 'V', then <i>Q</i> of the Schur vectors is updated. If <i>compq</i> = 'N', then no Schur vectors are updated.
<i>select</i>	LOGICAL. Array, DIMENSION at least max (1, <i>n</i>). Specifies the eigenvalues in the selected cluster. To select an eigenvalue <i>lambda(j)</i> , <i>select(j)</i> must be .TRUE.. <i>For real flavors:</i> to select a complex conjugate pair of eigenvalues <i>lambda(j)</i> and <i>lambda(j+1)</i> (corresponding 2 by 2 diagonal block), <i>select(j)</i> and/or <i>select(j+1)</i> must be .TRUE.; the complex conjugate <i>lambda(j)</i> and <i>lambda(j+1)</i> must be either both included in the cluster or both excluded.
<i>n</i>	INTEGER. The order of the matrix <i>T</i> (<i>n</i> \geq 0).
<i>t, q, work</i>	REAL for strsen DOUBLE PRECISION for dtrsen COMPLEX for ctrsen DOUBLE COMPLEX for ztrsen. Arrays: <i>t (ldt,*)</i> The <i>n</i> -by- <i>n</i> <i>T</i> . The second dimension of <i>t</i> must be at least max(1, <i>n</i>). <i>q (ldq,*)</i> If <i>compq</i> = 'V', then <i>q</i> must contain <i>Q</i> of Schur vectors. If <i>compq</i> = 'N', then <i>q</i> is not referenced. The second dimension of <i>q</i> must be at least max(1, <i>n</i>) if <i>compq</i> = 'V' and at least 1 if <i>compq</i> = 'N'. <i>work</i> is a workspace array, its dimension max(1, <i>lwork</i>). <i>ldt</i> INTEGER. The leading dimension of <i>t</i> ; at least max(1, <i>n</i>).

<i>ldq</i>	INTEGER. The leading dimension of <i>q</i> ; If <i>compq</i> = 'N', then <i>ldq</i> ≥ 1 . If <i>compq</i> = 'V', then <i>ldq</i> $\geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . If <i>job</i> = 'V' or 'B', <i>lwork</i> $\geq \max(1, 2*m*(n-m))$. If <i>job</i> = 'E', then <i>lwork</i> $\geq \max(1, m*(n-m))$ If <i>job</i> = 'N', then <i>lwork</i> ≥ 1 for complex flavors and <i>lwork</i> $\geq \max(1, n)$ for real flavors. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> for details.
<i>iwork</i>	INTEGER. <i>iwork</i> (<i>liwork</i>) is a workspace array. The array <i>iwork</i> is not referenced if <i>job</i> = 'N' or 'E'. The actual amount of workspace required cannot exceed $n^2/2$ if <i>job</i> = 'V' or 'B'.
<i>liwork</i>	INTEGER. The dimension of the array <i>iwork</i> . If <i>job</i> = 'V' or 'B', <i>liwork</i> $\geq \max(1, 2m(n-m))$. If <i>job</i> = 'E' or 'E', <i>liwork</i> ≥ 1 . If <i>liwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>iwork</i> array, returns this value as the first entry of the <i>iwork</i> array, and no error message related to <i>liwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> for details.

Output Parameters

<i>t</i>	Overwritten by the updated matrix <i>R</i> .
<i>q</i>	If <i>compq</i> = 'V', <i>q</i> contains the updated matrix of Schur vectors; the first <i>m</i> columns of the <i>Q</i> form an orthogonal basis for the specified invariant subspace.
<i>w</i>	COMPLEX for <i>ctrsen</i> DOUBLE COMPLEX for <i>ztrsen</i> . Array, DIMENSION at least $\max(1, n)$. The recorded eigenvalues of <i>R</i> . The eigenvalues are stored in the same order as on the diagonal of <i>R</i> .
<i>wr, wi</i>	REAL for <i>strsen</i> DOUBLE PRECISION for <i>dtrsen</i>

Arrays, DIMENSION at least $\max(1, n)$. Contain the real and imaginary parts, respectively, of the reordered eigenvalues of R . The eigenvalues are stored in the same order as on the diagonal of R . Note that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

m

INTEGER.

For complex flavors: the number of the specified invariant subspaces, which is the same as the number of selected eigenvalues (see *select*).

For real flavors: the dimension of the specified invariant subspace. The value of *m* is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues (see *select*).

Constraint: $0 \leq m \leq n$.

s

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

If *job* = 'E' or 'B', *s* is a lower bound on the reciprocal condition number of the average of the selected cluster of eigenvalues.

If *m* = 0 or *n*, then *s* = 1.

For real flavors: if *info* = 1, then *s* is set to zero. *s* is not referenced if *job* = 'N' or 'V'.

sep

REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.

If *job* = 'V' or 'B', *sep* is the estimated reciprocal condition number of the specified invariant subspace.

If *m* = 0 or *n*, then *sep* = $|T|$.

For real flavors: if *info* = 1, then *sep* is set to zero.

sep is not referenced if *job* = 'N' or 'E'.

work(1)

On exit, if *info* = 0, then *work(1)* returns the optimal size of *lwork*.

iwork(1)

On exit, if *info* = 0, then *iwork(1)* returns the optimal size of *liwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the *i*-th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *trsen* interface are the following:

t

Holds the matrix T of size (n,n) .

select

Holds the vector of length *n*.

<i>wr</i>	Holds the vector of length n . Used in real flavors only.
<i>wi</i>	Holds the vector of length n . Used in real flavors only.
<i>w</i>	Holds the vector of length n . Used in complex flavors only.
<i>q</i>	Holds the matrix Q of size (n,n) .
<i>compq</i>	Restored based on the presence of the argument <i>q</i> as follows: <i>compq</i> = 'V', if <i>q</i> is present, <i>compq</i> = 'N', if <i>q</i> is omitted.
<i>job</i>	Restored based on the presence of arguments <i>s</i> and <i>sep</i> as follows: <i>job</i> = 'B', if both <i>s</i> and <i>sep</i> are present, <i>job</i> = 'E', if <i>s</i> is present and <i>sep</i> omitted, <i>job</i> = 'V', if <i>s</i> is omitted and <i>sep</i> present, <i>job</i> = 'N', if both <i>s</i> and <i>sep</i> are omitted.

Application Notes

The computed matrix R is exactly similar to a matrix $T+E$, where $\|E\|_2 = O(\varepsilon) * \|T\|_2$, and ε is the machine precision. The computed s cannot underestimate the true reciprocal condition number by more than a factor of $(\min(m, n-m))^{1/2}$; *sep* may differ from the true value by $(m^2 * n - m^2)^{1/2}$. The angle between the computed invariant subspace and the true subspace is $O(\varepsilon) * \|A\|_2 / \text{sep}$. Note that if a 2-by-2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2-by-2 block to break into two 1-by-1 blocks, that is, for a pair of complex eigenvalues to become purely real. The values of eigenvalues however are never changed by the re-ordering.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?trsyl

Solves Sylvester equation for real quasi-triangular or complex triangular matrices.

Syntax

FORTRAN 77:

```
call strsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call dtrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call ctrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
call ztrsyl(trana, tranb, isgn, m, n, a, lda, b, ldb, c, ldc, scale, info)
```

Fortran 95:

```
call trsyl(a, b, c, scale [, transa] [, transb] [, isgn] [, info])
```

C:

```
lapack_int LAPACKE_strsyl( int matrix_layout, char transa, char transb, lapack_int isgn,
                           lapack_int m, lapack_int n, const float* a, lapack_int lda, const float* b, lapack_int ldb,
                           float* c, lapack_int ldc, float* scale );

lapack_int LAPACKE_dtrsyl( int matrix_layout, char transa, char transb, lapack_int isgn,
                           lapack_int m, lapack_int n, const double* a, lapack_int lda, const double* b,
                           lapack_int ldb, double* c, lapack_int ldc, double* scale );

lapack_int LAPACKE_ctrsyl( int matrix_layout, char transa, char transb, lapack_int isgn,
                           lapack_int m, lapack_int n, const lapack_complex_float* a, lapack_int lda,
                           const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* c, lapack_int ldc,
                           float* scale );

lapack_int LAPACKE_ztrsyl( int matrix_layout, char transa, char transb, lapack_int isgn,
                           lapack_int m, lapack_int n, const lapack_complex_double* a, lapack_int lda,
                           const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* c, lapack_int ldc,
                           double* scale );
```

Include Files

- Fortran: mkl.fi
 - Fortran 95: lapack.f90
 - C: mkl.h

Description

The routine solves the Sylvester matrix equation $\text{op}(A) * X \pm X * \text{op}(B) = \alpha * C$, where $\text{op}(A) = A$ or A^H , and the matrices A and B are upper triangular (or, for real flavors, upper quasi-triangular in canonical Schur form); $\alpha \leq 1$ is a scale factor determined by the routine to avoid overflow in X ; A is m -by- m , B is n -by- n , and C and X are both m -by- n . The matrix X is obtained by a straightforward process of back substitution.

The equation has a unique solution if and only if $\alpha_i \pm \beta_i \neq 0$, where $\{\alpha_i\}$ and $\{\beta_i\}$ are the eigenvalues of A and B , respectively, and the sign (+ or -) is the same as that used in the equation to be solved.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

If $trans = 'N'$, then $op(A) = A$.

If $trans = 'T'$, then $\text{op}(A) = A^T$

If $\text{trans} = 'C'$ then $\text{op}(A) = A^H$.

CHARACTER^AT. Must Be N^A OR T^A OR C^A.

If $\text{trans} = \cdot N \cdot$, then $\text{op}(B) = B$.

If $transb = 'T'$, then $\text{op}(B) = B^\top$ (real flavors only).

If $trans = 'C'$, then $op(B) = B^H$.

<i>isgn</i>	INTEGER. Indicates the form of the Sylvester equation. If <i>isgn</i> = +1, $\text{op}(A) * X + X * \text{op}(B) = \text{alpha} * C$. If <i>isgn</i> = -1, $\text{op}(A) * X - X * \text{op}(B) = \text{alpha} * C$.
<i>m</i>	INTEGER. The order of <i>A</i> , and the number of rows in <i>X</i> and <i>C</i> (<i>m</i> ≥ 0).
<i>n</i>	INTEGER. The order of <i>B</i> , and the number of columns in <i>X</i> and <i>C</i> (<i>n</i> ≥ 0).
<i>a, b, c</i>	REAL for <i>strsyl</i> DOUBLE PRECISION for <i>dtrsyl</i> COMPLEX for <i>ctrsyl</i> DOUBLE COMPLEX for <i>ztrsyl</i> .
	Arrays: <i>a</i> (<i>lda</i> ,*) contains the matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, m)$. <i>b</i> (<i>ldb</i> ,*) contains the matrix <i>B</i> . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>c</i> (<i>ldc</i> ,*) contains the matrix <i>C</i> . The second dimension of <i>c</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; at least $\max(1, n)$.
<i>ldc</i>	INTEGER. The leading dimension of <i>c</i> ; at least $\max(1, n)$.

Output Parameters

<i>c</i>	Overwritten by the solution matrix <i>X</i> .
<i>scale</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. The value of the scale factor <i>α</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, <i>A</i> and <i>B</i> have common or close eigenvalues perturbed values were used to solve the equation.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *trsyl* interface are the following:

<i>a</i>	Holds the matrix A of size (m,m) .
<i>b</i>	Holds the matrix B of size (n,n) .
<i>c</i>	Holds the matrix C of size (m,n) .
<i>trana</i>	Must be ' <i>N</i> ', ' <i>C</i> ', or ' <i>T</i> '. The default value is ' <i>N</i> '.
<i>tranb</i>	Must be ' <i>N</i> ', ' <i>C</i> ', or ' <i>T</i> '. The default value is ' <i>N</i> '.
<i>isgn</i>	Must be +1 or -1. The default value is +1.

Application Notes

Let X be the exact, Y the corresponding computed solution, and R the residual matrix: $R = C - (AY \pm YB)$. Then the residual is always small:

$$\|R\|_F = O(\varepsilon) * (\|A\|_F + \|B\|_F) * \|Y\|_F.$$

However, Y is not necessarily the exact solution of a slightly perturbed equation; in other words, the solution is not backwards stable.

For the forward error, the following bound holds:

$$\|Y - X\|_F \leq \|R\|_F / \text{sep}(A, B)$$

but this may be a considerable overestimate. See [Golub96] for a definition of $\text{sep}(A, B)$.

The approximate number of floating-point operations for real flavors is $m^* n^* (m + n)$. For complex flavors it is 4 times greater.

Generalized Nonsymmetric Eigenvalue Problems

This section describes LAPACK routines for solving generalized nonsymmetric eigenvalue problems, reordering the generalized Schur factorization of a pair of matrices, as well as performing a number of related computational tasks.

A *generalized nonsymmetric eigenvalue problem* is as follows: given a pair of nonsymmetric (or non-Hermitian) n -by- n matrices A and B , find the *generalized eigenvalues* λ and the corresponding *generalized eigenvectors* x and y that satisfy the equations

$$Ax = \lambda Bx \quad (\text{right generalized eigenvectors } x)$$

and

$$y^H A = \lambda y^H B \quad (\text{left generalized eigenvectors } y).$$

Table "Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems" lists LAPACK routines (FORTRAN 77 interface) used to solve the generalized nonsymmetric eigenvalue problems and the generalized Sylvester equation. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems

Routine name	Operation performed
gghrd	Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.
ggbal	Balances a pair of general real or complex matrices.
ggbak	Forms the right or left eigenvectors of a generalized eigenvalue problem.
hgeqz	Implements the QZ method for finding the generalized eigenvalues of the matrix pair (H, T) .

Routine name	Operation performed
tgevc	Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices
tgexc	Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.
tgsen	Reorders the <i>generalized</i> Schur decomposition of a pair of matrices (A,B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A,B).
tgsyl	Solves the generalized Sylvester equation.
tgsyl	Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

?gghrd

Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.

Syntax**FORTRAN 77:**

```
call sgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call dgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call cgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
call zgghrd(compq, compz, n, ilo, ihi, a, lda, b, ldb, q, ldq, z, ldz, info)
```

Fortran 95:

```
call gghrd(a, b [,ilo] [,ihi] [,q] [,z] [,compq] [,compz] [,info])
```

C:

```
lapack_int LAPACKE_<?>gghrd( int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, <datatype>* a, lapack_int lda, <datatype>* b,
lapack_int ldb, <datatype>* q, lapack_int ldq, <datatype>* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reduces a pair of real/complex matrices (A, B) to generalized upper Hessenberg form using orthogonal/unitary transformations, where A is a general matrix and B is upper triangular. The form of the generalized eigenvalue problem is $A^*x = \lambda^*B^*x$, and B is typically made upper triangular by computing its QR factorization and moving the orthogonal matrix Q to the left side of the equation.

This routine simultaneously reduces A to a Hessenberg matrix H :

$$Q^H A^* Z = H$$

and transforms B to another upper triangular matrix T :

$$Q^H B^* Z = T$$

in order to reduce the problem to its standard form $H^*y = \lambda^*T^*y$, where $y = Z^H x$.

The orthogonal/unitary matrices Q and Z are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices Q_1 and Z_1 , so that

$$Q_1^* A^* Z_1^H = (Q_1^* Q)^* H^* (Z_1^* Z)^H$$

$$Q_1^* B^* Z_1^H = (Q_1^* Q)^* T^* (Z_1^* Z)^H$$

If Q_1 is the orthogonal/unitary matrix from the QR factorization of B in the original equation $A^* x = \lambda^* B^* x$, then the routine ?gghrd reduces the original problem to generalized Hessenberg form.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>compq</i>	CHARACTER*1. Must be 'N', 'I', or 'V'. If <i>compq</i> = 'N', matrix Q is not computed. If <i>compq</i> = 'I', Q is initialized to the unit matrix, and the orthogonal/unitary matrix Q is returned; If <i>compq</i> = 'V', Q must contain an orthogonal/unitary matrix Q_1 on entry, and the product $Q_1^* Q$ is returned.
<i>compz</i>	CHARACTER*1. Must be 'N', 'I', or 'V'. If <i>compz</i> = 'N', matrix Z is not computed. If <i>compz</i> = 'I', Z is initialized to the unit matrix, and the orthogonal/unitary matrix Z is returned; If <i>compz</i> = 'V', Z must contain an orthogonal/unitary matrix Z_1 on entry, and the product $Z_1^* Z$ is returned.
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. <i>ilo</i> and <i>ihi</i> mark the rows and columns of A which are to be reduced. It is assumed that A is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$. Values of <i>ilo</i> and <i>ihi</i> are normally set by a previous call to ggbal ; otherwise they should be set to 1 and n respectively. Constraint: If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, then <i>ilo</i> = 1 and <i>ihi</i> = 0.
<i>a, b, q, z</i>	REAL for sgghrd DOUBLE PRECISION for dgghrd COMPLEX for cgghrd DOUBLE COMPLEX for zgghrd. Arrays: <i>a</i> (<i>lda,*</i>) contains the n -by- n general matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb,*</i>) contains the n -by- n upper triangular matrix B . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>q</i> (<i>ldq,*</i>)

If $\text{compq} = \text{'N'}$, then q is not referenced.

If $\text{compq} = \text{'V'}$, then q must contain the orthogonal/unitary matrix Q_1 , typically from the QR factorization of B .

The second dimension of q must be at least $\max(1, n)$.

z ($ldz, *$)

If $\text{compz} = \text{'N'}$, then z is not referenced.

If $\text{compz} = \text{'V'}$, then z must contain the orthogonal/unitary matrix Z_1 .

The second dimension of z must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.

ldb INTEGER. The leading dimension of b ; at least $\max(1, n)$.

ldq INTEGER. The leading dimension of q ;

If $\text{compq} = \text{'N'}$, then $ldq \geq 1$.

If $\text{compq} = \text{'I'}$ or 'V' , then $ldq \geq \max(1, n)$.

ldz INTEGER. The leading dimension of z ;

If $\text{compz} = \text{'N'}$, then $ldz \geq 1$.

If $\text{compz} = \text{'I'}$ or 'V' , then $ldz \geq \max(1, n)$.

Output Parameters

a On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H , and the rest is set to zero.

b On exit, overwritten by the upper triangular matrix $T = Q^H * B * Z$. The elements below the diagonal are set to zero.

q If $\text{compq} = \text{'I'}$, then q contains the orthogonal/unitary matrix Q ;
If $\text{compq} = \text{'V'}$, then q is overwritten by the product $Q_1 * Q$.

z If $\text{compz} = \text{'I'}$, then z contains the orthogonal/unitary matrix Z ;
If $\text{compz} = \text{'V'}$, then z is overwritten by the product $Z_1 * Z$.

$info$ INTEGER.
If $info = 0$, the execution is successful.
If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gghrd` interface are the following:

a Holds the matrix A of size (n,n) .

<i>b</i>	Holds the matrix <i>B</i> of size (<i>n,n</i>).
<i>q</i>	Holds the matrix <i>Q</i> of size (<i>n,n</i>).
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n,n</i>).
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>compq</i>	If omitted, this argument is restored based on the presence of argument <i>q</i> as follows: <i>compq</i> = 'I', if <i>q</i> is present, <i>compq</i> = 'N', if <i>q</i> is omitted. If present, <i>compq</i> must be equal to 'I' or 'V' and the argument <i>q</i> must also be present. Note that there will be an error condition if <i>compq</i> is present and <i>q</i> omitted.
<i>compz</i>	If omitted, this argument is restored based on the presence of argument <i>z</i> as follows: <i>compz</i> = 'I', if <i>z</i> is present, <i>compz</i> = 'N', if <i>z</i> is omitted. If present, <i>compz</i> must be equal to 'I' or 'V' and the argument <i>z</i> must also be present. Note that there will be an error condition if <i>compz</i> is present and <i>z</i> omitted.

?ggbal*Balances a pair of general real or complex matrices.***Syntax****FORTRAN 77:**

```
call sggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call dggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call cggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
call zggbal(job, n, a, lda, b, ldb, ilo, ihi, lscale, rscale, work, info)
```

Fortran 95:

```
call ggbal(a, b [,ilo] [,ihi] [,lscale] [,rscale] [,job] [,info])
```

C:

```
lapack_int LAPACKE_sggbal( int matrix_layout, char job, lapack_int n, float* a,
lapack_int lda, float* b, lapack_int ldb, lapack_int* ilo, lapack_int* ihi, float*
lscale, float* rscale );

lapack_int LAPACKE_dggbal( int matrix_layout, char job, lapack_int n, double* a,
lapack_int lda, double* b, lapack_int ldb, lapack_int* ilo, lapack_int* ihi, double*
lscale, double* rscale );

lapack_int LAPACKE_cggbal( int matrix_layout, char job, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
lapack_int* ilo, lapack_int* ihi, float* lscale, float* rscale );

lapack_int LAPACKE_zggbal( int matrix_layout, char job, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
lapack_int* ilo, lapack_int* ihi, double* lscale, double* rscale );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine balances a pair of general real/complex matrices (A, B). This involves, first, permuting A and B by similarity transformations to isolate eigenvalues in the first 1 to $ilo-1$ and last $ihi+1$ to n elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ilo to ihi to make the rows and columns as close in norm as possible. Both steps are optional. Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem $A^*x = \lambda * B^*x$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>job</i>	CHARACTER*1. Specifies the operations to be performed on A and B . Must be 'N' or 'P' or 'S' or 'B'. If <i>job</i> = 'N', then no operations are done; simply set <i>ilo</i> =1, <i>ihi</i> = <i>n</i> , <i>lscale(i)</i> =1.0 and <i>rscale(i)</i> =1.0 for <i>i</i> = 1, ..., <i>n</i> . If <i>job</i> = 'P', then permute only. If <i>job</i> = 'S', then scale only. If <i>job</i> = 'B', then both permute and scale.
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>a, b</i>	REAL for sggbal DOUBLE PRECISION for dggbal COMPLEX for cggbal DOUBLE COMPLEX for zggbal. Arrays: <i>a(lda, *)</i> contains the matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b(ldb, *)</i> contains the matrix B . The second dimension of <i>b</i> must be at least $\max(1, n)$. If <i>job</i> = 'N', <i>a</i> and <i>b</i> are not referenced.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; at least $\max(1, n)$.
<i>work</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Workspace array, DIMENSION at least $\max(1, 6n)$ when $\text{job} = \text{'S'}$ or 'B' , or at least 1 when $\text{job} = \text{'N'}$ or 'P' .

Output Parameters

a, b	Overwritten by the balanced matrices A and B , respectively.
ilo, ihi	INTEGER. ilo and ihi are set to integers such that on exit $a(i,j)=0$ and $b(i,j)=0$ if $i > j$ and $j=1, \dots, ilo-1$ or $i=ihi+1, \dots, n$. If $\text{job} = \text{'N'}$ or 'S' , then $ilo = 1$ and $ihi = n$.
$lscale, rscale$	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least $\max(1, n)$. $lscale$ contains details of the permutations and scaling factors applied to the left side of A and B . If P_j is the index of the row interchanged with row j , and D_j is the scaling factor applied to row j , then $\begin{aligned} lscale(j) &= P_j, \text{ for } j = 1, \dots, ilo-1 \\ &= D_j, \text{ for } j = ilo, \dots, ihi \\ &= P_j, \text{ for } j = ihi+1, \dots, n. \end{aligned}$ $rscale$ contains details of the permutations and scaling factors applied to the right side of A and B . If P_j is the index of the column interchanged with column j , and D_j is the scaling factor applied to column j , then $\begin{aligned} rscale(j) &= P_j, \text{ for } j = 1, \dots, ilo-1 \\ &= D_j, \text{ for } j = ilo, \dots, ihi \\ &= P_j, \text{ for } j = ihi+1, \dots, n \end{aligned}$ The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggbal` interface are the following:

a	Holds the matrix A of size (n,n) .
b	Holds the matrix B of size (n,n) .
$lscale$	Holds the vector of length (n) .

<i>rscale</i>	Holds the vector of length (<i>n</i>).
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>job</i>	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.

?ggbak

Forms the right or left eigenvectors of a generalized eigenvalue problem.

Syntax**FORTRAN 77:**

```
call sggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call dggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call cggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
call zggbak(job, side, n, ilo, ihi, lscale, rscale, m, v, ldv, info)
```

Fortran 95:

```
call ggbak(v [, ilo] [, ihi] [, lscale] [, rscale] [, job] [, info])
```

C:

```
lapack_int LAPACKE_sggbak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const float* lscale, const float* rscale, lapack_int
m, float* v, lapack_int ldv );

lapack_int LAPACKE_dggbak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const double* lscale, const double* rscale, lapack_int
m, double* v, lapack_int ldv );

lapack_int LAPACKE_cggbak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const float* lscale, const float* rscale, lapack_int
m, lapack_complex_float* v, lapack_int ldv );

lapack_int LAPACKE_zggbak( int matrix_layout, char job, char side, lapack_int n,
lapack_int ilo, lapack_int ihi, const double* lscale, const double* rscale, lapack_int
m, lapack_complex_double* v, lapack_int ldv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine forms the right or left eigenvectors of a real/complex generalized eigenvalue problem

$$A^*x = \lambda B^*x$$

by backward transformation on the computed eigenvectors of the balanced pair of matrices output by [ggbal](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>job</i>	CHARACTER*1. Specifies the type of backward transformation required. Must be 'N', 'P', 'S', or 'B'. If <i>job</i> = 'N', then no operations are done; return. If <i>job</i> = 'P', then do backward transformation for permutation only. If <i>job</i> = 'S', then do backward transformation for scaling only. If <i>job</i> = 'B', then do backward transformation for both permutation and scaling. This argument must be the same as the argument <i>job</i> supplied to ?ggbal.
<i>side</i>	CHARACTER*1. Must be 'L' or 'R'. If <i>side</i> = 'L', then <i>v</i> contains left eigenvectors. If <i>side</i> = 'R', then <i>v</i> contains right eigenvectors.
<i>n</i>	INTEGER. The number of rows of the matrix <i>V</i> (<i>n</i> \geq 0).
<i>ilo, ihi</i>	INTEGER. The integers <i>ilo</i> and <i>ihi</i> determined by ?gbal. Constraint: If <i>n</i> > 0, then $1 \leq ilo \leq ihi \leq n$; if <i>n</i> = 0, then <i>ilo</i> = 1 and <i>ihi</i> = 0.
<i>lscale, rscale</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Arrays, DIMENSION at least max(1, <i>n</i>). The array <i>lscale</i> contains details of the permutations and/or scaling factors applied to the left side of <i>A</i> and <i>B</i> , as returned by ?ggbal. The array <i>rscale</i> contains details of the permutations and/or scaling factors applied to the right side of <i>A</i> and <i>B</i> , as returned by ?ggbal.
<i>m</i>	INTEGER. The number of columns of the matrix <i>V</i> (<i>m</i> \geq 0).
<i>v</i>	REAL for sggbak DOUBLE PRECISION for dggbak COMPLEX for cggbak DOUBLE COMPLEX for zggbak. Array <i>v</i> (<i>ldv</i> ,*). Contains the matrix of right or left eigenvectors to be transformed, as returned by tgevc. The second dimension of <i>v</i> must be at least max(1, <i>m</i>).
<i>ldv</i>	INTEGER. The leading dimension of <i>v</i> ; at least max(1, <i>n</i>).

Output Parameters

<i>v</i>	Overwritten by the transformed eigenvectors
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggbak` interface are the following:

<i>v</i>	Holds the matrix <i>V</i> of size (<i>n,m</i>).
<i>lscale</i>	Holds the vector of length <i>n</i> .
<i>rscale</i>	Holds the vector of length <i>n</i> .
<i>ilo</i>	Default value for this argument is <i>ilo</i> = 1.
<i>ihi</i>	Default value for this argument is <i>ihi</i> = <i>n</i> .
<i>job</i>	Must be 'B', 'S', 'P', or 'N'. The default value is 'B'.
<i>side</i>	If omitted, this argument is restored based on the presence of arguments <i>lscale</i> and <i>rscale</i> as follows: <i>side</i> = 'L', if <i>lscale</i> is present and <i>rscale</i> omitted, <i>side</i> = 'R', if <i>lscale</i> is omitted and <i>rscale</i> present. Note that there will be an error condition if both <i>lscale</i> and <i>rscale</i> are present or if they both are omitted.

?hgeqz

Implements the QZ method for finding the generalized eigenvalues of the matrix pair (*H,T*).

Syntax

FORTRAN 77:

```
call shgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphar, alphai, beta, q,
ldq, z, ldz, work, lwork, info)

call dhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alphar, alphai, beta, q,
ldq, z, ldz, work, lwork, info)

call chgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha, beta, q, ldq, z,
ldz, work, lwork, rwork, info)

call zhgeqz(job, compq, compz, n, ilo, ihi, h, ldh, t, ldt, alpha, beta, q, ldq, z,
ldz, work, lwork, rwork, info)
```

Fortran 95:

```
call hgeqz(h, t [,ilo] [,ihi] [,alphar] [,alphai] [,beta] [,q] [,z] [,job] [,compq]
[,compz] [,info])
```

```
call hgeqz(h, t [,ilo] [,ihi] [,alpha] [,beta] [,q] [,z] [,job] [,compq] [, compz]
[,info])
```

C:

```
lapack_int LAPACKE_shgeqz( int matrix_layout, char job, char compq, char compz,
lapack_int n, lapack_int ilo, lapack_int ihi, float* h, lapack_int ldh, float* t,
lapack_int ldt, float* alphar, float* alphai, float* beta, float* q, lapack_int ldq,
float* z, lapack_int ldz );
```

```
lapack_int LAPACKE_dhgeqz( int matrix_layout, char job, char compq, char compz,
lapack_int n, lapack_int ilo, lapack_int ihi, double* h, lapack_int ldh, double* t,
lapack_int ldt, double* alphar, double* alphai, double* beta, double* q, lapack_int ldq,
double* z, lapack_int ldz );
```

```
lapack_int LAPACKE_chgeqz( int matrix_layout, char job, char compq, char compz,
lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_float* h, lapack_int ldh,
lapack_complex_float* t, lapack_int ldt, lapack_complex_float* alpha,
lapack_complex_float* beta, lapack_complex_float* q, lapack_int ldq,
lapack_complex_float* z, lapack_int ldz );
```

```
lapack_int LAPACKE_zhgeqz( int matrix_layout, char job, char compq, char compz,
lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_double* h, lapack_int ldh,
lapack_complex_double* t, lapack_int ldt, lapack_complex_double* alpha,
lapack_complex_double* beta, lapack_complex_double* q, lapack_int ldq,
lapack_complex_double* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the eigenvalues of a real/complex matrix pair (H, T) , where H is an upper Hessenberg matrix and T is upper triangular, using the double-shift version (for real flavors) or single-shift version (for complex flavors) of the QZ method. Matrix pairs of this type are produced by the reduction to generalized upper Hessenberg form of a real/complex matrix pair (A, B) :

$$A = Q_1^* H^* Z_1^H, \quad B = Q_1^* T^* Z_1^H,$$

as computed by ?gghrd.

For real flavors:

If $job = 'S'$, then the Hessenberg-triangular pair (H, T) is also reduced to generalized Schur form,

$$H = Q^* S^* Z^T, \quad T = Q^* P^* Z^T,$$

where Q and Z are orthogonal matrices, P is an upper triangular matrix, and S is a quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks. The 1-by-1 blocks correspond to real eigenvalues of the matrix pair (H, T) and the 2-by-2 blocks correspond to complex conjugate pairs of eigenvalues.

Additionally, the 2-by-2 upper triangular diagonal blocks of P corresponding to 2-by-2 blocks of S are reduced to positive diagonal form, that is, if $S(j+1, j)$ is non-zero, then $P(j+1, j) = P(j, j+1) = 0$, $P(j, j) > 0$, and $P(j+1, j+1) > 0$.

For complex flavors:

If $job = 'S'$, then the Hessenberg-triangular pair (H, T) is also reduced to generalized Schur form,

$$H = Q^* S^* Z^H, T = Q^* P^* Z^H,$$

where Q and Z are unitary matrices, and S and P are upper triangular.

For all function flavors:

Optionally, the orthogonal/unitary matrix Q from the generalized Schur factorization may be postmultiplied into an input matrix Q_1 , and the orthogonal/unitary matrix Z may be postmultiplied into an input matrix Z_1 .

If Q_1 and Z_1 are the orthogonal/unitary matrices from ?gghrd that reduced the matrix pair (A,B) to generalized upper Hessenberg form, then the output matrices $Q_1 Q$ and $Z_1 Z$ are the orthogonal/unitary factors from the generalized Schur factorization of (A,B) :

$$A = (Q_1 Q)^* S^* (Z_1 Z)^H, B = (Q_1 Q)^* P^* (Z_1 Z)^H.$$

To avoid overflow, eigenvalues of the matrix pair (H,T) (equivalently, of (A,B)) are computed as a pair of values (α,β) . For chgeqz/zhgeqz, α and β are complex, and for shgeqz/dhgeqz, α is complex and β real. If β is nonzero, $\lambda = \alpha/\beta$ is an eigenvalue of the generalized nonsymmetric eigenvalue problem (GNEP)

$$A^* x = \lambda B^* x$$

and if α is nonzero, $\mu = \beta/\alpha$ is an eigenvalue of the alternate form of the GNEP

$$\mu^* A^* y = B^* y.$$

Real eigenvalues (for real flavors) or the values of α and β for the i -th eigenvalue (for complex flavors) can be read directly from the generalized Schur form:

$$\alpha = S(i,i), \beta = P(i,i).$$

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>job</i>	CHARACTER*1. Specifies the operations to be performed. Must be 'E' or 'S'. If <i>job</i> = 'E', then compute eigenvalues only; If <i>job</i> = 'S', then compute eigenvalues and the Schur form.
<i>compq</i>	CHARACTER*1. Must be 'N', 'I', or 'V'. If <i>compq</i> = 'N', left Schur vectors (<i>q</i>) are not computed; If <i>compq</i> = 'I', <i>q</i> is initialized to the unit matrix and the matrix of left Schur vectors of (H,T) is returned; If <i>compq</i> = 'V', <i>q</i> must contain an orthogonal/unitary matrix Q_1 on entry and the product $Q_1^* Q$ is returned.
<i>compz</i>	CHARACTER*1. Must be 'N', 'I', or 'V'. If <i>compz</i> = 'N', right Schur vectors (<i>z</i>) are not computed; If <i>compz</i> = 'I', <i>z</i> is initialized to the unit matrix and the matrix of right Schur vectors of (H,T) is returned; If <i>compz</i> = 'V', <i>z</i> must contain an orthogonal/unitary matrix Z_1 on entry and the product $Z_1^* Z$ is returned.

n INTEGER. The order of the matrices H , T , Q , and Z

$(n \geq 0)$.

ilo, ihi

INTEGER. *ilo* and *ihi* mark the rows and columns of *H* which are in Hessenberg form. It is assumed that *H* is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$.

Constraint:

If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;
if $n = 0$, then $ilo = 1$ and $ihi = 0$.

h, t, q, z, work

REAL for shgeqz
DOUBLE PRECISION for dhgeqz
COMPLEX for chgeqz
DOUBLE COMPLEX for zhgeqz.

Arrays:

On entry, *h*(*ldh,**) contains the n -by- n upper Hessenberg matrix *H*.

The second dimension of *h* must be at least $\max(1, n)$.

On entry, *t*(*ldt,**) contains the n -by- n upper triangular matrix *T*.

The second dimension of *t* must be at least $\max(1, n)$.

q (*ldq,**):

On entry, if *compq* = 'V', this array contains the orthogonal/unitary matrix Q_1 used in the reduction of (A,B) to generalized Hessenberg form.

If *compq* = 'N', then *q* is not referenced.

The second dimension of *q* must be at least $\max(1, n)$.

z (*ldz,**):

On entry, if *compz* = 'V', this array contains the orthogonal/unitary matrix Z_1 used in the reduction of (A,B) to generalized Hessenberg form.

If *compz* = 'N', then *z* is not referenced.

The second dimension of *z* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

ldh

INTEGER. The leading dimension of *h*; at least $\max(1, n)$.

ldt

INTEGER. The leading dimension of *t*; at least $\max(1, n)$.

ldq

INTEGER. The leading dimension of *q*;

If *compq* = 'N', then *ldq* ≥ 1 .

If *compq* = 'I' or 'V', then *ldq* $\geq \max(1, n)$.

ldz

INTEGER. The leading dimension of *z*;

If *compz* = 'N', then *ldz* ≥ 1 .

If *compz* = 'I' or 'V', then *ldz* $\geq \max(1, n)$.

lwork

INTEGER. The dimension of the array *work*.

lwork $\geq \max(1, n)$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by **xerbla**. See *Application Notes* for details.

rwork

REAL for chgeqz

DOUBLE PRECISION for zhgeqz.

Workspace array, DIMENSION at least $\max(1, n)$. Used in complex flavors only.

Output Parameters

h

For real flavors:

If *job* = 'S', then on exit *h* contains the upper quasi-triangular matrix *S* from the generalized Schur factorization.

If *job* = 'E', then on exit the diagonal blocks of *h* match those of *S*, but the rest of *h* is unspecified.

For complex flavors:

If *job* = 'S', then, on exit, *h* contains the upper triangular matrix *S* from the generalized Schur factorization.

If *job* = 'E', then on exit the diagonal of *h* matches that of *S*, but the rest of *h* is unspecified.

t

If *job* = 'S', then, on exit, *t* contains the upper triangular matrix *P* from the generalized Schur factorization.

For real flavors:

2-by-2 diagonal blocks of *P* corresponding to 2-by-2 blocks of *S* are reduced to positive diagonal form, that is, if *h*(*j*+1,*j*) is non-zero, then *t*(*j*+1, *j*)=*t*(*j*, *j*+1)=0 and *t*(*j*, *j*) and *t*(*j*+1, *j*+1) will be positive.

If *job* = 'E', then on exit the diagonal blocks of *t* match those of *P*, but the rest of *t* is unspecified.

For complex flavors:

if *job* = 'E', then on exit the diagonal of *t* matches that of *P*, but the rest of *t* is unspecified.

alphar, alphai

REAL for shgeqz;

DOUBLE PRECISION for dhgeqz.

Arrays, DIMENSION at least $\max(1, n)$. The real and imaginary parts, respectively, of each scalar *alpha* defining an eigenvalue of GNEP.

If *alphai*(*j*) is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j*+1)-th eigenvalues are a complex conjugate pair, with

alphai(*j*+1) = -*alphai*(*j*).

alpha

COMPLEX for chgeqz;

DOUBLE COMPLEX for zhgeqz.

Array, DIMENSION at least max(1, n).

The complex scalars *alpha* that define the eigenvalues of GNEP. *alphai(i) = S(i,i)* in the generalized Schur factorization.

beta

REAL for shgeqz
DOUBLE PRECISION for dhgeqz
COMPLEX for chgeqz
DOUBLE COMPLEX for zhgeqz.

Array, DIMENSION at least max(1, n).

For real flavors:

The scalars *beta* that define the eigenvalues of GNEP.

Together, the quantities *alpha = (alphar(j), alphai(j))* and *beta = beta(j)* represent the j-th eigenvalue of the matrix pair (A, B) , in one of the forms $\lambda = \alpha/\beta$ or $\mu = \beta/\alpha$. Since either λ or μ may overflow, they should not, in general, be computed.

For complex flavors:

The real non-negative scalars *beta* that define the eigenvalues of GNEP.

beta(i) = P(i,i) in the generalized Schur factorization. Together, the quantities *alpha = alpha(j)* and *beta = beta(j)* represent the j-th eigenvalue of the matrix pair (A, B) , in one of the forms $\lambda = \alpha/\beta$ or $\mu = \beta/\alpha$. Since either λ or μ may overflow, they should not, in general, be computed.

q

On exit, if *compq = 'I'*, *q* is overwritten by the orthogonal/unitary matrix of left Schur vectors of the pair (H, T) , and if *compq = 'V'*, *q* is overwritten by the orthogonal/unitary matrix of left Schur vectors of (A, B) .

z

On exit, if *compz = 'I'*, *z* is overwritten by the orthogonal/unitary matrix of right Schur vectors of the pair (H, T) , and if *compz = 'V'*, *z* is overwritten by the orthogonal/unitary matrix of right Schur vectors of (A, B) .

work(1)

If *info ≥ 0*, on exit, *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.

If *info = 0*, the execution is successful.

If *info = -i*, the *i*-th parameter had an illegal value.

If *info = 1, ..., n*, the QZ iteration did not converge.

(H, T) is not in Schur form, but *alphar(i)*, *alphai(i)* (for real flavors), *alpha(i)* (for complex flavors), and *beta(i)*, $i=info+1, \dots, n$ should be correct.

If *info = n+1, ..., 2n*, the shift calculation failed.

(H, T) is not in Schur form, but *alphar(i)*, *alphai(i)* (for real flavors), *alpha(i)* (for complex flavors), and *beta(i)*, $i = info-n+1, \dots, n$ should be correct.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hgeqz` interface are the following:

<i>h</i>	Holds the matrix H of size (n,n) .
<i>t</i>	Holds the matrix T of size (n,n) .
<i>alphar</i>	Holds the vector of length n . Used in real flavors only.
<i>alphai</i>	Holds the vector of length n . Used in real flavors only.
<i>alpha</i>	Holds the vector of length n . Used in complex flavors only.
<i>beta</i>	Holds the vector of length n .
<i>q</i>	Holds the matrix Q of size (n,n) .
<i>z</i>	Holds the matrix Z of size (n,n) .
<i>ilo</i>	Default value for this argument is $ilo = 1$.
<i>ihi</i>	Default value for this argument is $ihi = n$.
<i>job</i>	Must be 'E' or 'S'. The default value is 'E'.
<i>compq</i>	If omitted, this argument is restored based on the presence of argument <i>q</i> as follows: <i>compq</i> = 'I', if <i>q</i> is present, <i>compq</i> = 'N', if <i>q</i> is omitted. If present, <i>compq</i> must be equal to 'I' or 'V' and the argument <i>q</i> must also be present. Note that there will be an error condition if <i>compq</i> is present and <i>q</i> omitted.
<i>compz</i>	If omitted, this argument is restored based on the presence of argument <i>z</i> as follows: <i>compz</i> = 'I', if <i>z</i> is present, <i>compz</i> = 'N', if <i>z</i> is omitted. If present, <i>compz</i> must be equal to 'I' or 'V' and the argument <i>z</i> must also be present. Note an error condition if <i>compz</i> is present and <i>z</i> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set `lwork = -1`, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set `lwork` to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgevc

Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices.

Syntax

FORTRAN 77:

```
call stgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, info)
call dtgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, info)
call ctgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, rwork, info)
call ztgevc(side, howmny, select, n, s, lds, p, ldp, vl, ldvl, vr, ldvr, mm, m, work, rwork, info)
```

Fortran 95:

```
call tgevc(s, p [,howmny] [,select] [,vl] [,vr] [,m] [,info])
```

C:

```
lapack_int LAPACKE_<?>tgevc( int matrix_layout, char side, char howmny,
const lapack_logical* select, lapack_int n, const <datatype>* s, lapack_int lds,
const <datatype>* p, lapack_int ldp, <datatype>* vl, lapack_int ldvl, <datatype>* vr,
lapack_int ldvr, lapack_int mm, lapack_int* m );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes some or all of the right and/or left eigenvectors of a pair of real/complex matrices (S, P), where S is quasi-triangular (for real flavors) or upper triangular (for complex flavors) and P is upper triangular.

Matrix pairs of this type are produced by the generalized Schur factorization of a real/complex matrix pair (A, B):

$$A = Q^* S^* Z^H, B = Q^* P^* Z^H$$

as computed by ?gghrd plus ?hgeqz.

The right eigenvector x and the left eigenvector y of (S, P) corresponding to an eigenvalue w are defined by:

$$S^* x = w^* P^* x, Y^H S = w^* Y^H P$$

The eigenvalues are not input to this routine, but are computed directly from the diagonal blocks or diagonal elements of S and P .

This routine returns the matrices X and/or Y of right and left eigenvectors of (S,P) , or the products Z^*X and/or Q^*Y , where Z and Q are input matrices.

If Q and Z are the orthogonal/unitary factors from the generalized Schur factorization of a matrix pair (A,B) , then Z^*X and Q^*Y are the matrices of right and left eigenvectors of (A,B) .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>side</i>	CHARACTER*1. Must be 'R', 'L', or 'B'. If <i>side</i> = 'R', compute right eigenvectors only. If <i>side</i> = 'L', compute left eigenvectors only. If <i>side</i> = 'B', compute both right and left eigenvectors.
<i>howmny</i>	CHARACTER*1. Must be 'A', 'B', or 'S'. If <i>howmny</i> = 'A', compute all right and/or left eigenvectors. If <i>howmny</i> = 'B', compute all right and/or left eigenvectors, backtransformed by the matrices in <i>vr</i> and/or <i>vl</i> . If <i>howmny</i> = 'S', compute selected right and/or left eigenvectors, specified by the logical array <i>select</i> .
<i>select</i>	LOGICAL. Array, DIMENSION at least max (1, <i>n</i>). If <i>howmny</i> = 'S', <i>select</i> specifies the eigenvectors to be computed. If <i>howmny</i> = 'A' or 'B', <i>select</i> is not referenced. <i>For real flavors:</i> If <i>omega(j)</i> is a real eigenvalue, the corresponding real eigenvector is computed if <i>select(j)</i> is .TRUE.. If <i>omega(j)</i> and <i>omega(j+1)</i> are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either <i>select(j)</i> or <i>select(j+1)</i> is .TRUE., and on exit <i>select(j)</i> is set to .TRUE. and <i>select(j+1)</i> is set to .FALSE.. <i>For complex flavors:</i> The eigenvector corresponding to the <i>j</i> -th eigenvalue is computed if <i>select(j)</i> is .TRUE.. <i>n</i>
<i>s, p, vl, vr, work</i>	REAL for stgevc DOUBLE PRECISION for dtgevc COMPLEX for ctgevc DOUBLE COMPLEX for ztgevc. Arrays:

$s(lsd,*)$ contains the matrix S from a generalized Schur factorization as computed by `?hgeqz`. This matrix is upper quasi-triangular for real flavors, and upper triangular for complex flavors.

The second dimension of s must be at least $\max(1, n)$.

$p(lap,*)$ contains the upper triangular matrix P from a generalized Schur factorization as computed by `?hgeqz`.

For real flavors, 2-by-2 diagonal blocks of P corresponding to 2-by-2 blocks of S must be in positive diagonal form.

For complex flavors, P must have real diagonal elements. The second dimension of p must be at least $\max(1, n)$.

If $side = 'L'$ or $'B'$ and $howmny = 'B'$, $vl(ldvl,*)$ must contain an n -by- n matrix Q (usually the orthogonal/unitary matrix Q of left Schur vectors returned by `?hgeqz`). The second dimension of vl must be at least $\max(1, mm)$.

If $side = 'R'$, vl is not referenced.

If $side = 'R'$ or $'B'$ and $howmny = 'B'$, $vr(ldrv,*)$ must contain an n -by- n matrix Z (usually the orthogonal/unitary matrix Z of right Schur vectors returned by `?hgeqz`). The second dimension of vr must be at least $\max(1, mm)$.

If $side = 'L'$, vr is not referenced.

$work(*)$ is a workspace array.

DIMENSION at least $\max(1, 6*n)$ for real flavors and at least $\max(1, 2*n)$ for complex flavors.

lds

INTEGER. The leading dimension of s ; at least $\max(1, n)$.

ldp

INTEGER. The leading dimension of p ; at least $\max(1, n)$.

$ldvl$

INTEGER. The leading dimension of vl ;

If $side = 'L'$ or $'B'$, then $ldvl \geq n$.

If $side = 'R'$, then $ldvl \geq 1$.

$ldvr$

INTEGER. The leading dimension of vr ;

If $side = 'R'$ or $'B'$, then $ldvr \geq n$.

If $side = 'L'$, then $ldvr \geq 1$.

mm

INTEGER. The number of columns in the arrays vl and/or vr ($mm \geq m$).

$rwork$

REAL for `ctgevc` DOUBLE PRECISION for `ztgevc`. Workspace array, DIMENSION at least $\max(1, 2*n)$. Used in complex flavors only.

Output Parameters

vl

On exit, if $side = 'L'$ or $'B'$, vl contains:

if $howmny = 'A'$, the matrix Y of left eigenvectors of (S,P) ;

if $howmny = 'B'$, the matrix $Q*Y$;

if *howmny* = 'S', the left eigenvectors of (*S,P*) specified by *select*, stored consecutively in the columns of *vl*, in the same order as their eigenvalues.

For real flavors:

A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

vr

On exit, if *side* = 'R' or 'B', *vr* contains:

if *howmny* = 'A', the matrix *X* of right eigenvectors of (*S,P*);

if *howmny* = 'B', the matrix *Z*X*;

if *howmny* = 'S', the right eigenvectors of (*S,P*) specified by *select*, stored consecutively in the columns of *vr*, in the same order as their eigenvalues.

For real flavors:

A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

m

INTEGER. The number of columns in the arrays *vl* and/or *vr* actually used to store the eigenvectors.

If *howmny* = 'A' or 'B', *m* is set to *n*.

For real flavors:

Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.

For complex flavors:

Each selected eigenvector occupies one column.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

For real flavors:

if *info* = *i*>0, the 2-by-2 block (*i*:*i*+1) does not have a complex eigenvalue.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tgevc` interface are the following:

s Holds the matrix *S* of size (*n,n*).

p Holds the matrix *P* of size (*n,n*).

select Holds the vector of length *n*.

vl Holds the matrix *VL* of size (*n,mm*).

<i>vr</i>	Holds the matrix <i>VR</i> of size (<i>n,mm</i>).
<i>side</i>	Restored based on the presence of arguments <i>vl</i> and <i>vr</i> as follows: <i>side</i> = ' <i>B</i> ', if both <i>vl</i> and <i>vr</i> are present, <i>side</i> = ' <i>L</i> ', if <i>vl</i> is present and <i>vr</i> omitted, <i>side</i> = ' <i>R</i> ', if <i>vl</i> is omitted and <i>vr</i> present, Note that there will be an error condition if both <i>vl</i> and <i>vr</i> are omitted.
<i>howmny</i>	If omitted, this argument is restored based on the presence of argument <i>select</i> as follows: <i>howmny</i> = ' <i>S</i> ', if <i>select</i> is present, <i>howmny</i> = ' <i>A</i> ', if <i>select</i> is omitted. If present, <i>howmny</i> must be equal to ' <i>A</i> ' or ' <i>B</i> ' and the argument <i>select</i> must be omitted. Note that there will be an error condition if both <i>howmny</i> and <i>select</i> are present.

?tgexc

Reorders the generalized Schur decomposition of a pair of matrices (A,B) so that one diagonal block of (A,B) moves to another row index.

Syntax**FORTRAN 77:**

```
call stgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, work, lwork,
info)

call dtgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, work, lwork,
info)

call ctgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, info)

call ztgexc(wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, ifst, ilst, info)
```

Fortran 95:

```
call tgexc(a, b [,ifst] [,ilst] [,z] [,q] [,info])
```

C:

```
lapack_int LAPACKE_<?>tgexc( int matrix_layout, lapack_logical wantq, lapack_logical
wantz, lapack_int n, <datatype>* a, lapack_int lda, <datatype>* b, lapack_int ldb,
<datatype>* q, lapack_int ldq, <datatype>* z, lapack_int ldz, lapack_int* ifst,
lapack_int* ilst );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (*A,B*) using an orthogonal/unitary equivalence transformation

$$(A, B) = Q^* (A, B) * Z^H,$$

so that the diagonal block of (A, B) with row index $ifst$ is moved to row $ilst$. Matrix pair (A, B) must be in a generalized real-Schur/Schur canonical form (as returned by [gges](#)), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks and B is upper triangular. Optionally, the matrices Q and Z of generalized Schur vectors are updated.

$$Q(\text{in}) * A(\text{in}) * Z(\text{in})' = Q(\text{out}) * A(\text{out}) * Z(\text{out})'$$

$$Q(\text{in}) * B(\text{in}) * Z(\text{in})' = Q(\text{out}) * B(\text{out}) * Z(\text{out})'.$$

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>wantq, wantz</i>	LOGICAL. If <i>wantq</i> = .TRUE., update the left transformation matrix Q ; If <i>wantq</i> = .FALSE., do not update Q ; If <i>wantz</i> = .TRUE., update the right transformation matrix Z ; If <i>wantz</i> = .FALSE., do not update Z .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>a, b, q, z</i>	REAL for <i>stgexc</i> DOUBLE PRECISION for <i>dtgexc</i> COMPLEX for <i>ctgexc</i> DOUBLE COMPLEX for <i>ztgexc</i> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb</i> ,*) contains the matrix B . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>q</i> (<i>ldq</i> ,*) If <i>wantq</i> = .FALSE., then <i>q</i> is not referenced. If <i>wantq</i> = .TRUE., then <i>q</i> must contain the orthogonal/unitary matrix Q . The second dimension of <i>q</i> must be at least $\max(1, n)$. <i>z</i> (<i>ldz</i> ,*) If <i>wantz</i> = .FALSE., then <i>z</i> is not referenced. If <i>wantz</i> = .TRUE., then <i>z</i> must contain the orthogonal/unitary matrix Z . The second dimension of <i>z</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; at least $\max(1, n)$.
<i>ldq</i>	INTEGER. The leading dimension of <i>q</i> ; If <i>wantq</i> = .FALSE., then <i>ldq</i> ≥ 1 .

	If <code>wantq</code> = .TRUE., then $ldq \geq \max(1, n)$.
<code>ldz</code>	INTEGER. The leading dimension of Z ; If <code>wantz</code> = .FALSE., then $ldz \geq 1$. If <code>wantz</code> = .TRUE., then $ldz \geq \max(1, n)$.
<code>ifst, ilst</code>	INTEGER. Specify the reordering of the diagonal blocks of (A, B) . The block with row index <code>ifst</code> is moved to row <code>ilst</code> , by a sequence of swapping between adjacent blocks. Constraint: $1 \leq ifst, ilst \leq n$.
<code>work</code>	REAL for <code>stgexc</code> ; DOUBLE PRECISION for <code>dtgexc</code> . Workspace array, DIMENSION ($/work$). Used in real flavors only.
<code>lwork</code>	INTEGER. The dimension of <code>work</code> ; must be at least $4n + 16$. If <code>lwork</code> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <code>work</code> array, returns this value as the first entry of the <code>work</code> array, and no error message related to <code>lwork</code> is issued by <code>xerbla</code> . See <i>Application Notes</i> for details.

Output Parameters

<code>a, b, q, z</code>	Overwritten by the updated matrices A, B, Q , and Z respectively.
<code>ifst, ilst</code>	Overwritten for real flavors only. If <code>ifst</code> pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; <code>ilst</code> always points to the first row of the block in its final position (which may differ from its input value by ± 1).
<code>info</code>	INTEGER. If <code>info</code> = 0, the execution is successful. If <code>info</code> = $-i$, the i -th parameter had an illegal value. If <code>info</code> = 1, the transformed matrix pair (A, B) would be too far from generalized Schur form; the problem is ill-conditioned. (A, B) may have been partially reordered, and <code>ilst</code> points to the first row of the current position of the block being moved.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tgexc` interface are the following:

<code>a</code>	Holds the matrix A of size (n,n) .
<code>b</code>	Holds the matrix B of size (n,n) .
<code>z</code>	Holds the matrix Z of size (n,n) .
<code>q</code>	Holds the matrix Q of size (n,n) .

<i>wantq</i>	Restored based on the presence of the argument <i>q</i> as follows: <i>wantq</i> = .TRUE., if <i>q</i> is present, <i>wantq</i> = .FALSE., if <i>q</i> is omitted.
<i>wantz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>wantz</i> = .TRUE., if <i>z</i> is present, <i>wantz</i> = .FALSE., if <i>z</i> is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgse

Reorders the generalized Schur decomposition of a pair of matrices (*A,B*) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (*A,B*).

Syntax

FORTRAN 77:

```
call stgse(ijob, wantq, wantz, select, n, a, lda, b, ldb, alphar, alphai, beta, q,
ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)
call dtgse(ijob, wantq, wantz, select, n, a, lda, b, ldb, alphar, alphai, beta, q,
ldq, z, ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)
call ctgse(ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha, beta, q, ldq, z,
ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)
call ztgse(ijob, wantq, wantz, select, n, a, lda, b, ldb, alpha, beta, q, ldq, z,
ldz, m, pl, pr, dif, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call tgse(a, b, select [,alphar] [,alphai] [,beta] [,ijob] [,q] [,z] [,pl] [,pr] [,dif]
[,m] [,info])
call tgse(a, b, select [,alpha] [,beta] [,ijob] [,q] [,z] [,pl] [,pr] [,dif] [,m]
[,info])
```

C:

```
lapack_int LAPACKE_stgse( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n, float* a, lapack_int
lda, float* b, lapack_int ldb, float* alphar, float* alphai, float* beta, float* q,
lapack_int ldq, float* z, lapack_int ldz, lapack_int* m, float* pl, float* pr, float*
dif );
```

```

lapack_int LAPACKE_dtgsen( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n, double* a, lapack_int lda,
double* b, lapack_int ldb, double* alphar, double* alphai, double* beta, double* q,
lapack_int ldq, double* z, lapack_int ldz, lapack_int* m, double* pl, double* pr,
double* dif );

lapack_int LAPACKE_ctgsen( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n, lapack_complex_float* a,
lapack_int lda, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* alpha,
lapack_complex_float* beta, lapack_complex_float* q, lapack_int ldq,
lapack_complex_float* z, lapack_int ldz, lapack_int* m, float* pl, float* pr, float*
dif );

lapack_int LAPACKE_ztgsen( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* alpha, lapack_complex_double* beta, lapack_complex_double* q,
lapack_int ldq, lapack_complex_double* z, lapack_int ldz, lapack_int* m, double* pl,
double* pr, double* dif );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair (A, B) (in terms of an orthogonal/unitary equivalence transformation $Q^T (A, B) Z$ for real flavors or $Q^H (A, B) Z$ for complex flavors), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair (A, B) . The leading columns of Q and Z form orthonormal/unitary bases of the corresponding left and right eigenspaces (deflating subspaces).

(A, B) must be in generalized real-Schur/Schur canonical form (as returned by [gges](#)), that is, A and B are both upper triangular.

?tgse also computes the generalized eigenvalues

$\omega_j = (\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$ (for real flavors)

$\omega_j = \text{alpha}(j)/\text{beta}(j)$ (for complex flavors)

of the reordered matrix pair (A, B) .

Optionally, the routine computes the estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are $\text{Difu}[(A_{11}, B_{11}), (A_{22}, B_{22})]$ and $\text{Difl}[(A_{11}, B_{11}), (A_{22}, B_{22})]$, that is, the separation(s) between the matrix pairs (A_{11}, B_{11}) and (A_{22}, B_{22}) that correspond to the selected cluster and the eigenvalues outside the cluster, respectively, and norms of "projections" onto left and right eigenspaces with respect to the selected cluster in the (1,1)-block.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>ijob</i>	INTEGER. Specifies whether condition numbers are required for the cluster of eigenvalues (<i>pl</i> and <i>pr</i>) or the deflating subspaces <i>Difu</i> and <i>Difl</i> .
-------------	---

If $i_{job} = 0$, only reorder with respect to $select$;
 If $i_{job} = 1$, reciprocal of norms of "projections" onto left and right eigenspaces with respect to the selected cluster (pl and pr);
 If $i_{job} = 2$, compute upper bounds on Dif_u and Dif_l , using F-norm-based estimate ($dif(1:2)$);
 If $i_{job} = 3$, compute estimate of Dif_u and Dif_l , sing 1-norm-based estimate ($dif(1:2)$). This option is about 5 times as expensive as $i_{job} = 2$;
 If $i_{job} = 4$, compute pl , pr and dif (i.e., options 0, 1 and 2 above). This is an economic version to get it all;
 If $i_{job} = 5$, compute pl , pr and dif (i.e., options 0, 1 and 3 above).

$wantq, wantz$

LOGICAL.

If $wantq = .TRUE.$, update the left transformation matrix Q ;
 If $wantq = .FALSE.$, do not update Q ;
 If $wantz = .TRUE.$, update the right transformation matrix Z ;
 If $wantz = .FALSE.$, do not update Z .

$select$

LOGICAL.

Array, DIMENSION at least $\max(1, n)$. Specifies the eigenvalues in the selected cluster.

To select an eigenvalue $\omega(j)$, $select(j)$ must be $.TRUE.$. For real flavors: to select a complex conjugate pair of eigenvalues $\omega(j)$ and $\omega(j+1)$ (corresponding 2 by 2 diagonal block), $select(j)$ and/or $select(j+1)$ must be set to $.TRUE.$; the complex conjugate $\omega(j)$ and $\omega(j+1)$ must be either both included in the cluster or both excluded.

n

INTEGER. The order of the matrices A and B ($n \geq 0$).

$a, b, q, z, work$

REAL for $stgse$

DOUBLE PRECISION for $dtgse$

COMPLEX for $ctgse$

DOUBLE COMPLEX for $ztgse$.

Arrays:

$a(l_{da}, *)$ contains the matrix A .

For real flavors: A is upper quasi-triangular, with (A, B) in generalized real Schur canonical form.

For complex flavors: A is upper triangular, in generalized Schur canonical form.

The second dimension of a must be at least $\max(1, n)$.

$b(l_{db}, *)$ contains the matrix B .

For real flavors: B is upper triangular, with (A, B) in generalized real Schur canonical form.

For complex flavors: B is upper triangular, in generalized Schur canonical form. The second dimension of b must be at least $\max(1, n)$.

q (ldq,)*

If *wantq* = .TRUE., then *q* is an *n*-by-*n* matrix;
 If *wantq* = .FALSE., then *q* is not referenced.
 The second dimension of *q* must be at least $\max(1, n)$.

z (ldz,)*

If *wantz* = .TRUE., then *z* is an *n*-by-*n* matrix;
 If *wantz* = .FALSE., then *z* is not referenced.
 The second dimension of *z* must be at least $\max(1, n)$.
work is a workspace array, its dimension $\max(1, lwork)$.

lda
 INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

ldb
 INTEGER. The leading dimension of *b*; at least $\max(1, n)$.

ldq
 INTEGER. The leading dimension of *q*; *ldq* ≥ 1 .
 If *wantq* = .TRUE., then *ldq* $\geq \max(1, n)$.

ldz
 INTEGER. The leading dimension of *z*; *ldz* ≥ 1 .
 If *wantz* = .TRUE., then *ldz* $\geq \max(1, n)$.

lwork
 INTEGER. The dimension of the array *work*.
For real flavors:
 If *ijob* = 1, 2, or 4, *lwork* $\geq \max(4n+16, 2m(n-m))$.
 If *ijob* = 3 or 5, *lwork* $\geq \max(4n+16, 4m(n-m))$.
For complex flavors:
 If *ijob* = 1, 2, or 4, *lwork* $\geq \max(1, 2m(n-m))$.
 If *ijob* = 3 or 5, *lwork* $\geq \max(1, 4m(n-m))$.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*. See *Application Notes* for details.

iwork
 INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork
 INTEGER. The dimension of the array *iwork*.
For real flavors:
 If *ijob* = 1, 2, or 4, *liwork* $\geq n+6$.
 If *ijob* = 3 or 5, *liwork* $\geq \max(n+6, 2m(n-m))$.
For complex flavors:
 If *ijob* = 1, 2, or 4, *liwork* $\geq n+2$.
 If *ijob* = 3 or 5, *liwork* $\geq \max(n+2, 2m(n-m))$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $iwork$ array, returns this value as the first entry of the $iwork$ array, and no error message related to $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

a, b	Overwritten by the reordered matrices A and B , respectively.
$alphar, alphai$	<p>REAL for <code>stgSEN</code>;</p> <p>DOUBLE PRECISION for <code>dtgSEN</code>.</p> <p>Arrays, DIMENSION at least $\max(1, n)$. Contain values that form generalized eigenvalues in real flavors.</p> <p>See <i>beta</i>.</p>
$alpha$	<p>COMPLEX for <code>ctgSEN</code>;</p> <p>DOUBLE COMPLEX for <code>ztgSEN</code>.</p> <p>Array, DIMENSION at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors.</p> <p>See <i>beta</i>.</p>
$beta$	<p>REAL for <code>stgSEN</code></p> <p>DOUBLE PRECISION for <code>dtgSEN</code></p> <p>COMPLEX for <code>ctgSEN</code></p> <p>DOUBLE COMPLEX for <code>ztgSEN</code>.</p> <p>Array, DIMENSION at least $\max(1, n)$.</p> <p><i>For real flavors:</i></p> <p>On exit, $(alphar(j) + alphai(j)*i)/beta(j)$, $j=1, \dots, n$, will be the generalized eigenvalues.</p> <p>$alphar(j) + alphai(j)*i$ and $beta(j)$, $j=1, \dots, n$ are the diagonals of the complex Schur form (S, T) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (A, B) were further reduced to triangular form using complex unitary transformations.</p> <p>If $alphai(j)$ is zero, then the j-th eigenvalue is real; if positive, then the j-th and $(j+1)$-st eigenvalues are a complex conjugate pair, with $alphai(j+1)$ negative.</p> <p><i>For complex flavors:</i></p> <p>The diagonal elements of A and B, respectively, when the pair (A, B) has been reduced to generalized Schur form. $alpha(i)/beta(i)$, $i=1, \dots, n$ are the generalized eigenvalues.</p> <p>If $wantq = .TRUE.$, then, on exit, Q has been postmultiplied by the left orthogonal transformation matrix which reorder (A, B). The leading m columns of Q form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).</p>

z If *wantz* = .TRUE., then, on exit, *Z* has been postmultiplied by the left orthogonal transformation matrix which reorder (*A*, *B*). The leading *m* columns of *Z* form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).

m INTEGER.
 The dimension of the specified pair of left and right eigen-spaces (deflating subspaces); $0 \leq m \leq n$.

pl, pr REAL for single precision flavors;
 DOUBLE PRECISION for double precision flavors.
 If *ijob* = 1, 4, or 5, *pl* and *pr* are lower bounds on the reciprocal of the norm of "projections" onto left and right eigenspaces with respect to the selected cluster.
 $0 < pl, pr \leq 1$. If *m* = 0 or *m* = *n*, *pl* = *pr* = 1.
 If *ijob* = 0, 2 or 3, *pl* and *pr* are not referenced

dif REAL for single precision flavors;DOUBLE PRECISION for double precision flavors.
 Array, DIMENSION (2).
 If *ijob* $\geq 2, *dif(1:2)* store the estimates of *Difu* and *Difl*.
 If *ijob* = 2 or 4, *dif(1:2)* are F-norm-based upper bounds on *Difu* and *Difl*.
 If *ijob* = 3 or 5, *dif(1:2)* are 1-norm-based estimates of *Difu* and *Difl*.
 If *m* = 0 or *n*, *dif(1:2)* = F-norm([*A*, *B*]).
 If *ijob* = 0 or 1, *dif* is not referenced.$

work(1) If *ijob* is not 0 and *info* = 0, on exit, *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

iwork(1) If *ijob* is not 0 and *info* = 0, on exit, *iwork(1)* contains the minimum value of *liwork* required for optimum performance. Use this *liwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = $-i$, the *i*-th parameter had an illegal value.
 If *info* = 1, Reordering of (*A*, *B*) failed because the transformed matrix pair (*A*, *B*) would be too far from generalized Schur form; the problem is very ill-conditioned. (*A*, *B*) may have been partially reordered.
 If requested, 0 is returned in *dif(*)*, *pl* and *pr*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tgsen` interface are the following:

<i>a</i>	Holds the matrix A of size (n,n) .
<i>b</i>	Holds the matrix B of size (n,n) .
<i>select</i>	Holds the vector of length n .
<i>alphar</i>	Holds the vector of length n . Used in real flavors only.
<i>alphai</i>	Holds the vector of length n . Used in real flavors only.
<i>alpha</i>	Holds the vector of length n . Used in complex flavors only.
<i>beta</i>	Holds the vector of length n .
<i>q</i>	Holds the matrix Q of size (n,n) .
<i>z</i>	Holds the matrix Z of size (n,n) .
<i>dif</i>	Holds the vector of length (2).
<i>ijob</i>	Must be 0, 1, 2, 3, 4, or 5. The default value is 0.
<i>wantq</i>	Restored based on the presence of the argument <i>q</i> as follows: <i>wantq</i> = .TRUE., if <i>q</i> is present, <i>wantq</i> = .FALSE., if <i>q</i> is omitted.
<i>wantz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>wantz</i> = .TRUE., if <i>z</i> is present, <i>wantz</i> = .FALSE., if <i>z</i> is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsvl

Solves the generalized Sylvester equation.

Syntax

FORTRAN 77:

```
call stgsvl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale,
dif, work, lwork, iwork, info)
```

```
call dtgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale,
dif, work, lwork, iwork, info)

call ctgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale,
dif, work, lwork, iwork, info)

call ztgsyl(trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf, scale,
dif, work, lwork, iwork, info)
```

Fortran 95:

```
call tgsyl(a, b, c, d, e, f [,ijob] [,trans] [,scale] [,dif] [,info])
```

C:

```
lapack_int LAPACKE_stgsyl( int matrix_layout, char trans, lapack_int ijob, lapack_int
m, lapack_int n, const float* a, lapack_int lda, const float* b, lapack_int ldb,
float* c, lapack_int ldc, const float* d, lapack_int ldd, const float* e, lapack_int
lde, float* f, lapack_int ldf, float* scale, float* dif );

lapack_int LAPACKE_dtgsyl( int matrix_layout, char trans, lapack_int ijob, lapack_int
m, lapack_int n, const double* a, lapack_int lda, const double* b, lapack_int ldb,
double* c, lapack_int ldc, const double* d, lapack_int ldd, const double* e,
lapack_int lde, double* f, lapack_int ldf, double* scale, double* dif );

lapack_int LAPACKE_ctgsyl( int matrix_layout, char trans, lapack_int ijob, lapack_int
m, lapack_int n, const lapack_complex_float* a, lapack_int lda,
const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* c, lapack_int ldc,
const lapack_complex_float* d, lapack_int ldd, const lapack_complex_float* e,
lapack_int lde, lapack_complex_float* f, lapack_int ldf, float* scale, float* dif );

lapack_int LAPACKE_ztgsyl( int matrix_layout, char trans, lapack_int ijob, lapack_int
m, lapack_int n, const lapack_complex_double* a, lapack_int lda,
const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* c, lapack_int ldc,
const lapack_complex_double* d, lapack_int ldd, const lapack_complex_double* e,
lapack_int lde, lapack_complex_double* f, lapack_int ldf, double* scale, double* dif );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves the generalized Sylvester equation:

$$A^*R - L^*B = \text{scale}^*C$$

$$D^*R - L^*E = \text{scale}^*F$$

where R and L are unknown m -by- n matrices, (A, D) , (B, E) and (C, F) are given matrix pairs of size m -by- m , n -by- n and m -by- n , respectively, with real/complex entries. (A, D) and (B, E) must be in generalized real-Schur/Schur canonical form, that is, A, B are upper quasi-triangular/triangular and D, E are upper triangular.

The solution (R, L) overwrites (C, F) . The factor scale , $0 \leq \text{scale} \leq 1$, is an output scaling factor chosen to avoid overflow.

In matrix notation the above equation is equivalent to the following: solve $Z^*x = \text{scale}^*b$, where Z is defined as

$$Z = \begin{pmatrix} \text{kron}(I_n, A) - \text{kron}(B^T, I_m) \\ \text{kron}(I_n, D) - \text{kron}(E^T, I_m) \end{pmatrix}$$

Here I_k is the identity matrix of size k and X' is the transpose/conjugate-transpose of X . $\text{kron}(X, Y)$ is the Kronecker product between the matrices X and Y .

If $\text{trans} = 'T'$ (for real flavors), or $\text{trans} = 'C'$ (for complex flavors), the routine `?tgssyl` solves the transposed/conjugate-transposed system $Z' * y = \text{scale} * b$, which is equivalent to solve for R and L in

$$A'^* R + D'^* L = \text{scale} * C$$

$$R^* B^* + L^* E^* = \text{scale} * (-F)$$

This case ($\text{trans} = 'T'$ for `stgssyl/dtgssyl` or $\text{trans} = 'C'$ for `ctgssyl/ztgssyl`) is used to compute an one-norm-based estimate of $\text{Dif}[(A, D), (B, E)]$, the separation between the matrix pairs (A, D) and (B, E) , using `lacon/lacon`.

If $i\text{job} \geq 1$, `?tgssyl` computes a Frobenius norm-based estimate of $\text{Dif}[(A, D), (B, E)]$. That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of Z . This is a level 3 BLAS algorithm.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>trans</code>	CHARACTER*1. Must be ' <code>N</code> ', ' <code>T</code> ', or ' <code>C</code> '.
	If <code>trans = 'N'</code> , solve the generalized Sylvester equation.
	If <code>trans = 'T'</code> , solve the 'transposed' system (for real flavors only).
	If <code>trans = 'C'</code> , solve the 'conjugate transposed' system (for complex flavors only).
<code>ijob</code>	INTEGER. Specifies what kind of functionality to be performed:
	If <code>ijob = 0</code> , solve the generalized Sylvester equation only;
	If <code>ijob = 1</code> , perform the functionality of <code>ijob = 0</code> and <code>ijob = 3</code> ;
	If <code>ijob = 2</code> , perform the functionality of <code>ijob = 0</code> and <code>ijob = 4</code> ;
	If <code>ijob = 3</code> , only an estimate of $\text{Dif}[(A, D), (B, E)]$ is computed (look ahead strategy is used);
	If <code>ijob = 4</code> , only an estimate of $\text{Dif}[(A, D), (B, E)]$ is computed (?gecon on sub-systems is used). If <code>trans = 'T'</code> or ' <code>C</code> ', <code>ijob</code> is not referenced.
<code>m</code>	INTEGER. The order of the matrices A and D , and the row dimension of the matrices C, F, R and L .
<code>n</code>	INTEGER. The order of the matrices B and E , and the column dimension of the matrices C, F, R and L .
<code>a, b, c, d, e, f, work</code>	REAL for <code>stgssyl</code> DOUBLE PRECISION for <code>dtgssyl</code> COMPLEX for <code>ctgssyl</code>

DOUBLE COMPLEX for `ztgsyl`.

Arrays:

`a(lda,*)` contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix A .

The second dimension of `a` must be at least $\max(1, m)$.

`b ldb,*)` contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix B . The second dimension of `b` must be at least $\max(1, n)$.

`c ldc,*)` contains the right-hand-side of the first matrix equation in the generalized Sylvester equation (as defined by `trans`)

The second dimension of `c` must be at least $\max(1, n)$.

`d ldd,*)` contains the upper triangular matrix D .

The second dimension of `d` must be at least $\max(1, m)$.

`e lde,*)` contains the upper triangular matrix E .

The second dimension of `e` must be at least $\max(1, n)$.

`f ldf,*)` contains the right-hand-side of the second matrix equation in the generalized Sylvester equation (as defined by `trans`)

The second dimension of `f` must be at least $\max(1, n)$.

`work` is a workspace array, its dimension $\max(1, \text{lwork})$.

`lda`

INTEGER. The leading dimension of `a`; at least $\max(1, m)$.

`ldb`

INTEGER. The leading dimension of `b`; at least $\max(1, n)$.

`ldc`

INTEGER. The leading dimension of `c`; at least $\max(1, m)$.

`lld`

INTEGER. The leading dimension of `d`; at least $\max(1, m)$.

`lde`

INTEGER. The leading dimension of `e`; at least $\max(1, n)$.

`ldf`

INTEGER. The leading dimension of `f`; at least $\max(1, m)$.

`lwork`

INTEGER.

The dimension of the array `work`. $\text{lwork} \geq 1$.

If `ijob = 1` or `2` and `trans = 'N'`, $\text{lwork} \geq \max(1, 2*m*n)$.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`. See *Application Notes* for details.

`iwork`

INTEGER. Workspace array, DIMENSION at least $(m+n+6)$ for real flavors, and at least $(m+n+2)$ for complex flavors.

Output Parameters

`c`

If `ijob=0, 1, or 2`, overwritten by the solution R .

If $i_{job}=3$ or 4 and $trans = 'N'$, c holds R , the solution achieved during the computation of the Dif-estimate.

f

If $i_{job}=0$, 1 , or 2 , overwritten by the solution L .

If $i_{job}=3$ or 4 and $trans = 'N'$, f holds L , the solution achieved during the computation of the Dif-estimate.

dif

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

On exit, dif is the reciprocal of a lower bound of the reciprocal of the Dif-function, that is, dif is an upper bound of $Dif[(A, D), (B, E)] = \sigma_{\min}(Z)$, where Z as in (2).

If $i_{job} = 0$, or $trans = 'T'$ (for real flavors), or $trans = 'C'$ (for complex flavors), dif is not touched.

scale

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

On exit, $scale$ is the scaling factor in the generalized Sylvester equation.

If $0 < scale < 1$, c and f hold the solutions R and L , respectively, to a slightly perturbed system but the input matrices A , B , D and E have not been changed.

If $scale = 0$, c and f hold the solutions R and L , respectively, to the homogeneous system with $C = F = 0$. Normally, $scale = 1$.

work(1)

If $info = 0$, $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info > 0$, (A, D) and (B, E) have common or close eigenvalues.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tgsyl` interface are the following:

a Holds the matrix A of size (m,m) .

b Holds the matrix B of size (n,n) .

c Holds the matrix C of size (m,n) .

d Holds the matrix D of size (m,m) .

e Holds the matrix E of size (n,n) .

f Holds the matrix F of size (m,n) .

<i>ijob</i>	Must be 0, 1, 2, 3, or 4. The default value is 0.
<i>trans</i>	Must be 'N' or 'T'. The default value is 'N'.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?tgsna

Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

Syntax

FORTRAN 77:

```
call stgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm, m, work, lwork, iwork, info)
call dtgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm, m, work, lwork, iwork, info)
call ctgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm, m, work, lwork, iwork, info)
call ztgsna(job, howmny, select, n, a, lda, b, ldb, vl, ldvl, vr, ldvr, s, dif, mm, m, work, lwork, iwork, info)
```

Fortran 95:

```
call tgsna(a, b [,s] [,dif] [,vl] [,vr] [,select] [,m] [,info])
```

C:

```
lapack_int LAPACKE_stgsna( int matrix_layout, char job, char howmny,
const lapack_logical* select, lapack_int n, const float* a, lapack_int lda,
const float* b, lapack_int ldb, const float* vl, lapack_int ldvl, const float* vr,
lapack_int ldvr, float* s, float* dif, lapack_int mm, lapack_int* m );
lapack_int LAPACKE_dtgsna( int matrix_layout, char job, char howmny,
const lapack_logical* select, lapack_int n, const double* a, lapack_int lda,
const double* b, lapack_int ldb, const double* vl, lapack_int ldvl, const double* vr,
lapack_int ldvr, double* s, double* dif, lapack_int mm, lapack_int* m );
lapack_int LAPACKE_ctgsna( int matrix_layout, char job, char howmny,
const lapack_logical* select, lapack_int n, const lapack_complex_float* a, lapack_int lda,
const lapack_complex_float* b, lapack_int ldb, const lapack_complex_float* vl,
lapack_int ldvl, const lapack_complex_float* vr, lapack_int ldvr, float* s, float*
dif, lapack_int mm, lapack_int* m );
```

```
lapack_int LAPACKE_ztgsna( int matrix_layout, char job, char howmny,
const lapack_logical* select, lapack_int n, const lapack_complex_double* a, lapack_int lda,
const lapack_complex_double* b, lapack_int ldb, const lapack_complex_double* vl,
lapack_int ldvl, const lapack_complex_double* vr, lapack_int ldvr, double* s, double* dif,
lapack_int mm, lapack_int* m );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The real flavors stgsna/dtgsna of this routine estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any matrix pair ($Q^*A^*Z^T, Q^*B^*Z^T$) with orthogonal matrices Q and Z .

(A, B) must be in generalized real Schur form (as returned by [gges/gges](#)), that is, A is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

The complex flavors ctgsna/ztgsna estimate reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a matrix pair (A, B). (A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>job</i>	CHARACTER*1. Specifies whether condition numbers are required for eigenvalues or eigenvectors. Must be 'E' or 'V' or 'B'. If <i>job</i> = 'E', for eigenvalues only (compute <i>s</i>). If <i>job</i> = 'V', for eigenvectors only (compute <i>dif</i>). If <i>job</i> = 'B', for both eigenvalues and eigenvectors (compute both <i>s</i> and <i>dif</i>).
<i>howmny</i>	CHARACTER*1. Must be 'A' or 'S'. If <i>howmny</i> = 'A', compute condition numbers for all eigenpairs. If <i>howmny</i> = 'S', compute condition numbers for selected eigenpairs specified by the logical array <i>select</i> .
<i>select</i>	LOGICAL. Array, DIMENSION at least max (1, <i>n</i>). If <i>howmny</i> = 'S', <i>select</i> specifies the eigenpairs for which condition numbers are required. If <i>howmny</i> = 'A', <i>select</i> is not referenced. <i>For real flavors:</i> To select condition numbers for the eigenpair corresponding to a real eigenvalue $\omega(j)$, <i>select(j)</i> must be set to .TRUE.; to select condition numbers corresponding to a complex conjugate pair of eigenvalues $\omega(j)$ and $\omega(j+1)$, either <i>select(j)</i> or <i>select(j+1)</i> must be set to .TRUE..

For complex flavors:

To select condition numbers for the corresponding j -th eigenvalue and/or eigenvector, *select(j)* must be set to .TRUE..

n

INTEGER. The order of the square matrix pair (A, B)
($n \geq 0$).

a, b, vl, vr, work

REAL for stgsna
DOUBLE PRECISION for dtgsna
COMPLEX for ctgsna
DOUBLE COMPLEX for ztgsna.

Arrays:

a(*lda,**) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix A in the pair (A, B).

The second dimension of *a* must be at least $\max(1, n)$.

b(*ldb,**) contains the upper triangular matrix B in the pair (A, B). The second dimension of *b* must be at least $\max(1, n)$.

If *job* = 'E' or 'B', *vl*(*ldvl,**) must contain left eigenvectors of (A, B), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vl*, as returned by ?tgevc.

If *job* = 'V', *vl* is not referenced. The second dimension of *vl* must be at least $\max(1, m)$.

If *job* = 'E' or 'B', *vr*(*ldvr,**) must contain right eigenvectors of (A, B), corresponding to the eigenpairs specified by *howmny* and *select*. The eigenvectors must be stored in consecutive columns of *vr*, as returned by ?tgevc.

If *job* = 'V', *vr* is not referenced. The second dimension of *vr* must be at least $\max(1, m)$.

work is a workspace array, its dimension $\max(1, lwork)$.

If *job* = 'E', *work* is not referenced.

lda

INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

ldb

INTEGER. The leading dimension of *b*; at least $\max(1, n)$.

ldvl

INTEGER. The leading dimension of *vl*; *ldvl* ≥ 1 .

If *job* = 'E' or 'B', then *ldvl* $\geq \max(1, n)$.

ldvr

INTEGER. The leading dimension of *vr*; *ldvr* ≥ 1 .

If *job* = 'E' or 'B', then *ldvr* $\geq \max(1, n)$.

mm

INTEGER. The number of elements in the arrays *s* and *dif* (*mm* $\geq m$).

lwork

INTEGER. The dimension of the array *work*.

lwork $\geq \max(1, n)$.

If $\text{job} = \text{'V'}$ or 'B' , $\text{lwork} \geq 2*n*(n+2)+16$ for real flavors, and $\text{lwork} \geq \max(1, 2*n*n)$ for complex flavors.

If $\text{lwork} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the work array, returns this value as the first entry of the work array, and no error message related to lwork is issued by [xerbla](#). See *Application Notes* for details.

$iwork$

INTEGER. Workspace array, DIMENSION at least $(n+6)$ for real flavors, and at least $(n+2)$ for complex flavors.

If $\text{job} = \text{'E'}$, $iwork$ is not referenced.

Output Parameters

s

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, DIMENSION (mm) .

If $\text{job} = \text{'E'}$ or 'B' , contains the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array.

If $\text{job} = \text{'V'}$, s is not referenced.

For real flavors:

For a complex conjugate pair of eigenvalues two consecutive elements of s are set to the same value. Thus, $s(j)$, $\text{dif}(j)$, and the j -th columns of v_l and v_r all correspond to the same eigenpair (but not in general the j -th eigenpair, unless all eigenpairs are selected).

dif

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, DIMENSION (mm) .

If $\text{job} = \text{'V'}$ or 'B' , contains the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array.

If the eigenvalues cannot be reordered to compute $\text{dif}(j)$, $\text{dif}(j)$ is set to 0; this can only occur when the true value would be very small anyway.

If $\text{job} = \text{'E'}$, dif is not referenced.

For real flavors:

For a complex eigenvector, two consecutive elements of dif are set to the same value.

For complex flavors:

For each eigenvalue/vector specified by select , dif stores a Frobenius norm-based estimate of Difl .

m

INTEGER. The number of elements in the arrays s and dif used to store the specified condition numbers; for each selected eigenvalue one element is used.

If $\text{howmny} = \text{'A'}$, m is set to n .

$\text{work}(1)$

$\text{work}(1)$

If job is not ' E ' and $info = 0$, on exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tgsna` interface are the following:

a	Holds the matrix A of size (n,n) .
b	Holds the matrix B of size (n,n) .
s	Holds the vector of length (mm) .
dif	Holds the vector of length (mm) .
vl	Holds the matrix VL of size (n,mm) .
vr	Holds the matrix VR of size (n,mm) .
$select$	Holds the vector of length n .
$howmny$	Restored based on the presence of the argument $select$ as follows: $howmny = 'S'$, if $select$ is present, $howmny = 'A'$, if $select$ is omitted.
job	Restored based on the presence of arguments s and dif as follows: $job = 'B'$, if both s and dif are present, $job = 'E'$, if s is present and dif omitted, $job = 'V'$, if s is omitted and dif present. Note that there will be an error condition if both s and dif are omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1 , then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Generalized Singular Value Decomposition

This section describes LAPACK computational routines used for finding the generalized singular value decomposition (GSVD) of two matrices A and B as

$$U^H A Q = D_1 * \begin{pmatrix} 0 & R \end{pmatrix},$$

$$V^H B Q = D_2 * (0 \quad R),$$

where U , V , and Q are orthogonal/unitary matrices, R is a nonsingular upper triangular matrix, and D_1 , D_2 are “diagonal” matrices of the structure detailed in the routines description section.

Table “Computational Routines for Generalized Singular Value Decomposition” lists LAPACK routines (FORTRAN 77 interface) that perform generalized singular value decomposition of matrices. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Generalized Singular Value Decomposition

Routine name	Operation performed
ggsvp	Computes the preprocessing decomposition for the generalized SVD
tgsja	Computes the generalized SVD of two upper triangular or trapezoidal matrices

You can use routines listed in the above table as well as the driver routine [ggsvd](#) to find the GSVD of a pair of general rectangular matrices.

?ggsvp

Computes the preprocessing decomposition for the generalized SVD.

Syntax

FORTRAN 77:

```
call sggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, tau, work, info)

call dggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, tau, work, info)

call cggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, rwork, tau, work, info)

call zggsvp(jobu, jobv, jobq, m, p, n, a, lda, b, ldb, tola, tolb, k, l, u, ldu, v,
ldv, q, ldq, iwork, rwork, tau, work, info)
```

Fortran 95:

```
call ggsvp(a, b, tola, tolb [, k] [,l] [,u] [,v] [,q] [,info])
```

C:

```
lapack_int LAPACKE_sggsvp( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, float* a, lapack_int lda, float* b,
lapack_int ldb, float tola, float tolb, lapack_int* k, lapack_int* l, float* u,
lapack_int ldu, float* v, lapack_int ldv, float* q, lapack_int ldq );

lapack_int LAPACKE_dggsvp( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, double* a, lapack_int lda, double* b,
lapack_int ldb, double tola, double tolb, lapack_int* k, lapack_int* l, double* u,
lapack_int ldu, double* v, lapack_int ldv, double* q, lapack_int ldq );

lapack_int LAPACKE_cggsvp( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_complex_float* a, lapack_int lda,
lapack_complex_float* b, lapack_int ldb, float tola, float tolb, lapack_int* k,
lapack_int* l, lapack_complex_float* u, lapack_int ldu, lapack_complex_float* v,
lapack_int ldv, lapack_complex_float* q, lapack_int ldq );
```

```
lapack_int LAPACKE_zggsrp( int matrix_layout, char jobu, char jobv, char jobq,
                           lapack_int m, lapack_int p, lapack_int n, lapack_complex_double* a, lapack_int lda,
                           lapack_complex_double* b, lapack_int ldb, double tola, double tolb, lapack_int* k,
                           lapack_int* l, lapack_complex_double* u, lapack_int ldu, lapack_complex_double* v,
                           lapack_int ldv, lapack_complex_double* q, lapack_int ldq );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes orthogonal matrices U , V and Q such that

$$\begin{matrix} \dots & \dots \\ \dots & \dots \end{matrix}$$

$$= \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \end{pmatrix}, \quad \text{if } m - k - l < 0$$

$$\begin{matrix} \dots & \dots \\ \dots & \dots \end{matrix}$$

where the k -by- k matrix A_{12} and l -by- l matrix B_{13} are nonsingular upper triangular; A_{23} is l -by- l upper triangular if $m - k - l \geq 0$, otherwise A_{23} is $(m-k)$ -by- l upper trapezoidal. The sum $k+l$ is equal to the effective numerical rank of the $(m+p)$ -by- n matrix $(A^H, B^H)^H$.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine [ggsrp](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobu</i>	CHARACTER*1. Must be 'U' or 'N'.
-------------	----------------------------------

If $\text{jobu} = \text{'U'}$, orthogonal/unitary matrix U is computed.

If $\text{jobu} = \text{'N'}$, U is not computed.

jobv

CHARACTER*1. Must be ' V ' or ' N '.

If $\text{jobv} = \text{'V'}$, orthogonal/unitary matrix V is computed.

If $\text{jobv} = \text{'N'}$, V is not computed.

jobq

CHARACTER*1. Must be ' Q ' or ' N '.

If $\text{jobq} = \text{'Q'}$, orthogonal/unitary matrix Q is computed.

If $\text{jobq} = \text{'N'}$, Q is not computed.

m

INTEGER. The number of rows of the matrix A ($m \geq 0$).

p

INTEGER. The number of rows of the matrix B ($p \geq 0$).

n

INTEGER. The number of columns of the matrices A and B ($n \geq 0$).

$a, b, tau, work$

REAL for sggsvp

DOUBLE PRECISION for dggsvp

COMPLEX for cggsvp

DOUBLE COMPLEX for zggsvp.

Arrays:

$a(l\text{da},*)$ contains the m -by- n matrix A .

The second dimension of a must be at least $\max(1, n)$.

$b(l\text{db},*)$ contains the p -by- n matrix B .

The second dimension of b must be at least $\max(1, n)$.

$tau(*)$ is a workspace array.

The dimension of tau must be at least $\max(1, n)$.

$work(*)$ is a workspace array.

The dimension of $work$ must be at least $\max(1, 3n, m, p)$.

$l\text{da}$

INTEGER. The leading dimension of a ; at least $\max(1, m)$.

$l\text{db}$

INTEGER. The leading dimension of b ; at least $\max(1, p)$.

$tola, tolb$

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

$tola$ and $tolb$ are the thresholds to determine the effective numerical rank of matrix B and a subblock of A . Generally, they are set to

$tola = \max(m, n) * ||A|| * \text{MACHEPS}$,

$tolb = \max(p, n) * ||B|| * \text{MACHEPS}$.

The size of $tola$ and $tolb$ may affect the size of backward errors of the decomposition.

<i>ldu</i>	INTEGER. The leading dimension of the output array u . $ldu \geq \max(1, m)$ if $jobu = 'U'$; $ldu \geq 1$ otherwise.
<i>ldv</i>	INTEGER. The leading dimension of the output array v . $ldv \geq \max(1, p)$ if $jobv = 'V'$; $ldv \geq 1$ otherwise.
<i>ldq</i>	INTEGER. The leading dimension of the output array q . $ldq \geq \max(1, n)$ if $jobq = 'Q'$; $ldq \geq 1$ otherwise.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <code>cggsvp</code> DOUBLE PRECISION for <code>zggsvp</code> . Workspace array, DIMENSION at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

<i>a</i>	Overwritten by the triangular (or trapezoidal) matrix described in the <i>Description</i> section.
<i>b</i>	Overwritten by the triangular matrix described in the <i>Description</i> section.
<i>k, l</i>	INTEGER. On exit, <i>k</i> and <i>l</i> specify the dimension of subblocks. The sum $k + l$ is equal to effective numerical rank of $(A^H, B^H)^H$.
<i>u, v, q</i>	REAL for <code>sggsvp</code> DOUBLE PRECISION for <code>dggsvp</code> COMPLEX for <code>cggsvp</code> DOUBLE COMPLEX for <code>zggsvp</code> . Arrays: If $jobu = 'U'$, $u(ldu,*)$ contains the orthogonal/unitary matrix U . The second dimension of u must be at least $\max(1, m)$. If $jobu = 'N'$, u is not referenced. If $jobv = 'V'$, $v(ldv,*)$ contains the orthogonal/unitary matrix V . The second dimension of v must be at least $\max(1, m)$. If $jobv = 'N'$, v is not referenced. If $jobq = 'Q'$, $q(ldq,*)$ contains the orthogonal/unitary matrix Q . The second dimension of q must be at least $\max(1, n)$. If $jobq = 'N'$, q is not referenced. <i>info</i> INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggsvp` interface are the following:

<i>a</i>	Holds the matrix A of size (m,n) .
<i>b</i>	Holds the matrix B of size (p,n) .
<i>u</i>	Holds the matrix U of size (m,m) .
<i>v</i>	Holds the matrix V of size (p,m) .
<i>q</i>	Holds the matrix Q of size (n,n) .
<i>jobu</i>	Restored based on the presence of the argument <i>u</i> as follows: <i>jobu</i> = 'U', if <i>u</i> is present, <i>jobu</i> = 'N', if <i>u</i> is omitted.
<i>jobv</i>	Restored based on the presence of the argument <i>v</i> as follows: <i>jobz</i> = 'V', if <i>v</i> is present, <i>jobz</i> = 'N', if <i>v</i> is omitted.
<i>jobq</i>	Restored based on the presence of the argument <i>q</i> as follows: <i>jobz</i> = 'Q', if <i>q</i> is present, <i>jobz</i> = 'N', if <i>q</i> is omitted.

?tgsja

Computes the generalized SVD of two upper triangular or trapezoidal matrices.

Syntax

FORTRAN 77:

```
call stgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha, beta,
u, ldu, v, ldv, q, ldq, work, ncycle, info)
call dtgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha, beta,
u, ldu, v, ldv, q, ldq, work, ncycle, info)
call ctgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha, beta,
u, ldu, v, ldv, q, ldq, work, ncycle, info)
call ztgsja(jobu, jobv, jobq, m, p, n, k, l, a, lda, b, ldb, tola, tolb, alpha, beta,
u, ldu, v, ldv, q, ldq, work, ncycle, info)
```

Fortran 95:

```
call tgsja(a, b, tola, tolb, k, l [,u] [,v] [,q] [,jobu] [,jobv] [,jobq] [,alpha]
[,beta] [,ncycle] [,info])
```

C:

```

lapack_int LAPACKE_stgsja( int matrix_layout, char jobu, char jobv, char jobq,
                           lapack_int m, lapack_int p, lapack_int n, lapack_int k, lapack_int l, float* a,
                           lapack_int lda, float* b, lapack_int ldb, float tola, float tolb, float* alpha, float*
                           beta, float* u, lapack_int ldu, float* v, lapack_int ldv, float* q, lapack_int ldq,
                           lapack_int* ncycle );

lapack_int LAPACKE_dtgsja( int matrix_layout, char jobu, char jobv, char jobq,
                           lapack_int m, lapack_int p, lapack_int n, lapack_int k, lapack_int l, double* a,
                           lapack_int lda, double* b, lapack_int ldb, double tola, double tolb, double* alpha,
                           double* beta, double* u, lapack_int ldu, double* v, lapack_int ldv, double* q,
                           lapack_int ldq, lapack_int* ncycle );

lapack_int LAPACKE_ctgsja( int matrix_layout, char jobu, char jobv, char jobq,
                           lapack_int m, lapack_int p, lapack_int n, lapack_int k, lapack_int l,
                           lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, float
                           tola, float tolb, float* alpha, float* beta, lapack_complex_float* u, lapack_int ldu,
                           lapack_complex_float* v, lapack_int ldv, lapack_complex_float* q, lapack_int ldq,
                           lapack_int* ncycle );

lapack_int LAPACKE_ztgsja( int matrix_layout, char jobu, char jobv, char jobq,
                           lapack_int m, lapack_int p, lapack_int n, lapack_int k, lapack_int l,
                           lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
                           double tola, double tolb, double* alpha, double* beta, lapack_complex_double* u,
                           lapack_int ldu, lapack_complex_double* v, lapack_int ldv, lapack_complex_double* q,
                           lapack_int ldq, lapack_int* ncycle );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the generalized singular value decomposition (GSVD) of two real/complex upper triangular (or trapezoidal) matrices A and B . On entry, it is assumed that matrices A and B have the following forms, which may be obtained by the preprocessing subroutine [ggsvp](#) from a general m -by- n matrix A and p -by- n matrix B :

$$A = \begin{matrix} & n - k - l & k & l \\ & k & \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix}, & \text{if } m - k - l \geq 0 \\ & l & & \\ m - k - l & & & \end{matrix}$$

.....
.....

...|!||||||| : : : : :

where the k -by- k matrix A_{12} and l -by- l matrix B_{13} are nonsingular upper triangular; A_{23} is l -by- l upper triangular if $m-k-l \geq 0$, otherwise A_{23} is $(m-k)$ -by- l upper trapezoidal.

On exit,

$$U^H A^* Q = D_1^* (0 \ R), \quad V^H B^* Q = D_2^* (0 \ R),$$

where U , V and Q are orthogonal/unitary matrices, R is a nonsingular upper triangular matrix, and D_1 and D_2 are "diagonal" matrices, which are of the following structures:

If $m-k-l \geq 0$,

$$D_1 = \begin{matrix} & k & l \\ \begin{matrix} & k \\ & l \\ m - k - l \end{matrix} & \begin{pmatrix} 0 & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{matrix}$$

...|!||||||| : : : :

$$(0 \ R) = \begin{matrix} n - k - l & k & l \\ \begin{matrix} & k \\ & l \end{matrix} & \begin{pmatrix} 0 & R_{11} & R_{12} \\ 0 & 0 & R_{22} \end{pmatrix} \end{matrix}$$

where

$$C = \text{diag}(\alpha(k+1), \dots, \alpha(k+1))$$

$$S = \text{diag}(\beta(k+1), \dots, \beta(k+1))$$

$$C^2 + S^2 = I$$

R is stored in $a(1:k+l, n-k-l+1:n)$ on exit.

If $m-k-l < 0$,

$$D_1 = \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix}$$

where

$C = \text{diag}(\alpha(K+1), \dots, \alpha(m))$,

```
S = diag(beta(K+1), ..., beta(m)),
```

$$C^2 + S^2 = I$$

$$\text{On exit} \quad \begin{pmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \end{pmatrix}$$

On exit, R_{22} and R_{23} are stored in $a(1:m, n-k-l+1:n)$ and R_{33} is stored in $b(m-k+1:l, n+m-k-l+1:n)$.

The computation of the orthogonal

either be formed explicitly, or they *may* be postmultiplied into input matrices U_1 , V_1 , or Q_1 .

The data types are

interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

CHARACTER MUST BE 0, 1, OR N.

If $\text{jobda} = 0$, a must contain an orthogonal/unitary matrix U_1 on entry.

If $\text{jobu} = \text{'T'}$, u is initialized to the unit matrix.

If $\text{Jobu} = \text{'N'}$, u is not computed.

jobv CHARACTER*1. Must be 'V', 'I', or 'N'.

If $\text{jobv} = \text{'V'}$, v must contain an orthogonal/unitary matrix V_1 on entry.

If $\text{jobv} = \text{'I'}$, v is initialized to the unit matrix.

If $\text{jobv} = \text{'N'}$, v is not computed.

jobq

CHARACTER*1. Must be ' Q ', ' I ', or ' N '.

If $\text{jobq} = \text{'Q'}$, q must contain an orthogonal/unitary matrix Q_1 on entry.

If $\text{jobq} = \text{'I'}$, q is initialized to the unit matrix.

If $\text{jobq} = \text{'N'}$, q is not computed.

m

INTEGER. The number of rows of the matrix A ($m \geq 0$).

p

INTEGER. The number of rows of the matrix B ($p \geq 0$).

n

INTEGER. The number of columns of the matrices A and B ($n \geq 0$).

k, l

INTEGER. Specify the subblocks in the input matrices A and B , whose GSVD is computed.

$a, b, u, v, q, work$

REAL for stgsja

DOUBLE PRECISION for dtgsja

COMPLEX for ctgsja

DOUBLE COMPLEX for ztgsja.

Arrays:

$a(lda,*)$ contains the m -by- n matrix A .

The second dimension of a must be at least $\max(1, n)$.

$b(lsb,*)$ contains the p -by- n matrix B .

The second dimension of b must be at least $\max(1, n)$.

If $\text{jobu} = \text{'U'}$, $u(ldu,*)$ must contain a matrix U_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).

The second dimension of u must be at least $\max(1, m)$.

If $\text{jobv} = \text{'V'}$, $v(ldv,*)$ must contain a matrix V_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).

The second dimension of v must be at least $\max(1, p)$.

If $\text{jobq} = \text{'Q'}$, $q(ldq,*)$ must contain a matrix Q_1 (usually the orthogonal/unitary matrix returned by ?ggsvp).

The second dimension of q must be at least $\max(1, n)$.

$work(*)$ is a workspace array.

The dimension of $work$ must be at least $\max(1, 2n)$.

lda

INTEGER. The leading dimension of a ; at least $\max(1, m)$.

lsb

INTEGER. The leading dimension of b ; at least $\max(1, p)$.

ldu

INTEGER. The leading dimension of the array u .

$lDU \geq \max(1, m)$ if $\text{jobu} = 'U'$; $lDU \geq 1$ otherwise.

ldv

INTEGER. The leading dimension of the array v .

$ldv \geq \max(1, p)$ if $\text{jobv} = 'V'$; $ldv \geq 1$ otherwise.

ldq

INTEGER. The leading dimension of the array q .

$ldq \geq \max(1, n)$ if $\text{jobq} = 'Q'$; $ldq \geq 1$ otherwise.

$tola, tolb$

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

$tola$ and $tolb$ are the convergence criteria for the Jacobi-Kogbetliantz iteration procedure. Generally, they are the same as used in ?ggsvp:

$tola = \max(m, n) * |A| * \text{MACHEPS}$,

$tolb = \max(p, n) * |B| * \text{MACHEPS}$.

Output Parameters

a

On exit, $a(n-k+1:n, 1:\min(k+l, m))$ contains the triangular matrix R or part of R .

b

On exit, if necessary, $b(m-k+1:l, n+m-k-l+1:n)$ contains a part of R .

$alpha, beta$

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays, DIMENSION at least $\max(1, n)$. Contain the generalized singular value pairs of A and B :

$alpha(1:k) = 1,$

$beta(1:k) = 0,$

and if $m-k-l \geq 0$,

$alpha(k+1:k+1) = \text{diag}(C),$

$beta(k+1:k+1) = \text{diag}(S),$

or if $m-k-l < 0$,

$alpha(k+1:m) = C, alpha(m+1:k+1) = 0$

$beta(K+1:M) = S,$

$beta(m+1:k+1) = 1.$

Furthermore, if $k+l < n$,

$alpha(k+l+1:n) = 0$ and

$beta(k+l+1:n) = 0.$

u

If $\text{jobu} = 'I'$, u contains the orthogonal/unitary matrix U .

If $\text{jobu} = 'U'$, u contains the product U_1^*U .

If $\text{jobu} = 'N'$, u is not referenced.

v

If $\text{jobv} = 'I'$, v contains the orthogonal/unitary matrix U .

	If $jobv = 'V'$, v contains the product V_1^*V .
	If $jobv = 'N'$, v is not referenced.
q	If $jobq = 'I'$, q contains the orthogonal/unitary matrix U .
	If $jobq = 'Q'$, q contains the product Q_1^*Q .
	If $jobq = 'N'$, q is not referenced.
$ncycle$	INTEGER. The number of cycles required for convergence.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = 1$, the procedure does not converge after MAXIT cycles.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `tgsja` interface are the following:

a	Holds the matrix A of size (m,n) .
b	Holds the matrix B of size (p,n) .
u	Holds the matrix U of size (m,m) .
v	Holds the matrix V of size (p,p) .
q	Holds the matrix Q of size (n,n) .
$alpha$	Holds the vector of length n .
$beta$	Holds the vector of length n .
$jobu$	If omitted, this argument is restored based on the presence of argument u as follows: $jobu = 'U'$, if u is present, $jobu = 'N'$, if u is omitted. If present, $jobu$ must be equal to ' I ' or ' U ' and the argument u must also be present. Note that there will be an error condition if $jobu$ is present and u omitted.
$jobv$	If omitted, this argument is restored based on the presence of argument v as follows: $jobv = 'V'$, if v is present, $jobv = 'N'$, if v is omitted. If present, $jobv$ must be equal to ' I ' or ' V ' and the argument v must also be present.

Note that there will be an error condition if $jobv$ is present and v omitted.

$jobq$ If omitted, this argument is restored based on the presence of argument q as follows:

$jobq = 'Q'$, if q is present,

$jobq = 'N'$, if q is omitted.

If present, $jobq$ must be equal to ' I ' or ' Q ' and the argument q must also be present.

Note that there will be an error condition if $jobq$ is present and q omitted.

Cosine-Sine Decomposition

This section describes LAPACK computational routines for computing the *cosine-sine decomposition* (CS decomposition) of a partitioned unitary/orthogonal matrix. The algorithm computes a complete 2-by-2 CS decomposition, which requires simultaneous diagonalization of all the four blocks of a unitary/orthogonal matrix partitioned into a 2-by-2 block structure.

The computation has the following phases:

1. The matrix is reduced to a bidiagonal block form.
2. The blocks are simultaneously diagonalized using techniques from the bidiagonal SVD algorithms.

Table "Computational Routines for Cosine-Sine Decomposition (CSD)" lists LAPACK routines (FORTRAN 77 interface) that perform CS decomposition of matrices. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Cosine-Sine Decomposition (CSD)

Operation	Real matrices	Complex matrices
Compute the CS decomposition of an orthogonal/unitary matrix in bidiagonal-block form	bbcisd/bbcisd	bbcisd/bbcisd
Simultaneously bidiagonalize the blocks of a partitioned orthogonal matrix	orbdb unbdb	
Simultaneously bidiagonalize the blocks of a partitioned unitary matrix		orbdb unbdb

See Also

[Cosine-Sine Decomposition](#)

?bbcisd

Computes the CS decomposition of an orthogonal/unitary matrix in bidiagonal-block form.

Syntax

FORTRAN 77:

```
call sbbcisd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b22e, work,
lwork, info )
```

```
call dbbcisd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b22e, work,
lwork, info )
```

```
call cbbcsd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b21e, b22e, rwork,
rlwork, info )

call zbbcsd( jobu1, jobu2, jobv1t, jobv2t, trans, m, p, q, theta, phi, u1, ldu1, u2,
ldu2, v1t, ldv1t, v2t, ldv2t, b11d, b11e, b12d, b12e, b21d, b21e, b21e, b22e, rwork,
rlwork, info )
```

Fortran 95:

```
call bbcsd( theta,phi,u1,u2,v1t,v2t[,b11d][,b11e][,b12d][,b12e][,b21d][,b21e][,b22d]
[,b22e][,jobu1][,jobu2][,jobv1t][,jobv2t][,trans][,info] )
```

C:

```
lapack_int LAPACKE_sbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, float* theta, float*
phi, float* u1, lapack_int ldu1, float* u2, lapack_int ldu2, float* v1t, lapack_int
ldv1t, float* v2t, lapack_int ldv2t, float* b11d, float* b11e, float* b12d, float*
b12e, float* b21d, float* b21e, float* b22d, float* b22e );

lapack_int LAPACKE_dbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, double* theta, double*
phi, double* u1, lapack_int ldu1, double* u2, lapack_int ldu2, double* v1t, lapack_int
ldv1t, double* v2t, lapack_int ldv2t, double* b11d, double* b11e, double* b12d,
double* b12e, double* b21d, double* b21e, double* b22d, double* b22e );

lapack_int LAPACKE_cbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, float* theta, float*
phi, lapack_complex_float* u1, lapack_int ldu1, lapack_complex_float* u2, lapack_int
ldu2, lapack_complex_float* v1t, lapack_int ldv1t, lapack_complex_float* v2t,
lapack_int ldv2t, float* b11d, float* b11e, float* b12d, float* b12e, float* b21d,
float* b21e, float* b22d, float* b22e );

lapack_int LAPACKE_zbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, double* theta, double*
phi, lapack_complex_double* u1, lapack_int ldu1, lapack_complex_double* u2, lapack_int
ldu2, lapack_complex_double* v1t, lapack_int ldv1t, lapack_complex_double* v2t,
lapack_int ldv2t, double* b11d, double* b11e, double* b12d, double* b12e, double*
b21d, double* b21e, double* b22d, double* b22e );
```

Include Files

- **Fortran:** mkl.fi
- **Fortran 95:** lapack.f90
- **C:** mkl.h

Description

`mkl_lapack.fi` The routine ?bbcsd computes the CS decomposition of an orthogonal or unitary matrix in bidiagonal-block form:

$$X = \begin{pmatrix} b_{11}|b_{12} & 0 & 0 \\ 0 | 0 & -I & 0 \\ b_{21}|b_{22} & 0 & 0 \\ 0 | 0 & 0 & I \end{pmatrix} = \left(\frac{u_1}{|} \frac{|}{u_2} \right) \begin{pmatrix} C & -S & 0 & 0 \\ 0 & 0 & -I & 0 \\ S & C & 0 & 0 \\ 0 & 0 & 0 & I \end{pmatrix} \left(\frac{v_1}{|} \frac{|}{v_2} \right)^T$$

or

$$X = \begin{pmatrix} b_{11}|b_{12} & 0 & 0 \\ 0 | 0 & -I & 0 \\ b_{21}|b_{22} & 0 & 0 \\ 0 | 0 & 0 & I \end{pmatrix} = \begin{pmatrix} u_1 & & \\ & \vdots & \\ & & u_2 \end{pmatrix} \begin{pmatrix} C & -S & 0 & 0 \\ 0 & 0 & -I & 0 \\ S & C & 0 & 0 \\ 0 & 0 & 0 & I \end{pmatrix} \begin{pmatrix} v_1 & & \\ & \vdots & \\ & & v_2 \end{pmatrix}^H$$

respectively.

x is m -by- m with the top-left block p -by- q . Note that q must not be larger than p , $m-p$, or $m-q$. If q is not the smallest index, x must be transposed and/or permuted in constant time using the `trans` option. See `?orcisd/?uncsd` for details.

The bidiagonal matrices b_{11} , b_{12} , b_{21} , and b_{22} are represented implicitly by angles `theta(1:q)` and `phi(1:q-1)`.

The orthogonal/unitary matrices u_1 , u_2 , v_1^t , and v_2^t are input/output. The input matrices are pre- or post-multiplied by the appropriate singular vector matrices.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>jobu1</code>	CHARACTER. If equals <code>Y</code> , then u_1 is updated. Otherwise, u_1 is not updated.
<code>jobu2</code>	CHARACTER. If equals <code>Y</code> , then u_2 is updated. Otherwise, u_2 is not updated.
<code>jobv1t</code>	CHARACTER. If equals <code>Y</code> , then v_1^t is updated. Otherwise, v_1^t is not updated.
<code>jobv2t</code>	CHARACTER. If equals <code>Y</code> , then v_2^t is updated. Otherwise, v_2^t is not updated.
<code>trans</code>	CHARACTER = <code>'T'</code> : $x, u_1, u_2, v_1^t, v_2^t$ are stored in row-major order. otherwise $x, u_1, u_2, v_1^t, v_2^t$ are stored in column-major order.
<code>m</code>	INTEGER. The number of rows and columns of the orthogonal/unitary matrix X in bidiagonal-block form.
<code>p</code>	INTEGER. The number of rows in the top-left block of x . $0 \leq p \leq m$.
<code>q</code>	INTEGER. The number of columns in the top-left block of x . $0 \leq q \leq \min(p, m-p, m-q)$.
<code>theta</code>	REAL for <code>sbbcsd</code> DOUBLE PRECISION for <code>dbbcsd</code> COMPLEX for <code>cbbcsd</code> DOUBLE COMPLEX for <code>zbbcsd</code> Array, DIMENSION (q).

On entry, the angles $\text{theta}(1), \dots, \text{theta}(q)$ that, along with $\text{phi}(1), \dots, \text{phi}(q-1)$, define the matrix in bidiagonal-block form.

phi REAL **for** sbbcsd

DOUBLE PRECISION **for** dbbcsd

COMPLEX **for** cbbcsd

DOUBLE COMPLEX **for** zbbcsd

Array, DIMENSION (q-1).

The angles $\text{phi}(1), \dots, \text{phi}(q-1)$ that, along with $\text{theta}(1), \dots, \text{theta}(q)$, define the matrix in bidiagonal-block form.

u1 REAL **for** sbbcsd

DOUBLE PRECISION **for** dbbcsd

COMPLEX **for** cbbcsd

DOUBLE COMPLEX **for** zbbcsd

Array, DIMENSION (ldu1,p).

On entry, an $ldu1$ -by- p matrix.

ldu1 INTEGER. The leading dimension of the array u_1 .

u2 REAL **for** sbbcsd

DOUBLE PRECISION **for** dbbcsd

COMPLEX **for** cbbcsd

DOUBLE COMPLEX **for** zbbcsd

Array, DIMENSION (ldu2,m-p).

On entry, an $ldu2$ -by-($m-p$) matrix.

ldu2 INTEGER. The leading dimension of the array u_2 .

v1t REAL **for** sbbcsd

DOUBLE PRECISION **for** dbbcsd

COMPLEX **for** cbbcsd

DOUBLE COMPLEX **for** zbbcsd

Array, DIMENSION (ldv1t,q).

On entry, an $ldv1t$ -by- q matrix.

ldv1t INTEGER. The leading dimension of the array $v1t$.

v2t REAL **for** sbbcsd

DOUBLE PRECISION **for** dbbcsd

COMPLEX **for** cbbcsd

DOUBLE COMPLEX **for** zbbcsd

Array, DIMENSION (ldv2t,m-q).

On entry, an $ldv2t$ -by- $(m-q)$ matrix.

$ldv2t$ INTEGER. The leading dimension of the array $v2t$.

$work$ REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Workspace array, DIMENSION ($\max(1, lwork)$).

$lwork$ INTEGER. The size of the $work$ array. $lwork \geq \max(1, 8*q)$

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla.

Output Parameters

$theta$ REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

On exit, the angles whose cosines and sines define the diagonal blocks in the CS decomposition.

$u1$ REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

On exit, $u1$ is postmultiplied by the left singular vector matrix common to $[b11 ; 0]$ and $[b12 \ 0 \ 0 ; 0 \ -I \ 0 \ 0]$.

$u2$ REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

On exit, $u2$ is postmultiplied by the left singular vector matrix common to $[b21 ; 0]$ and $[b22 \ 0 \ 0 ; 0 \ 0 \ I]$.

$v1t$ REAL for sbbcsd

DOUBLE PRECISION for dbbcsd

COMPLEX for cbbcsd

DOUBLE COMPLEX for zbbcsd

Array, DIMENSION (q).

On exit, $v1t$ is premultiplied by the transpose of the right singular vector matrix common to [$b11 ; 0$] and [$b21 ; 0$].

$v2t$ REAL **for** sbbcsd

DOUBLE PRECISION **for** dbbcsd

COMPLEX **for** cbbcsd

DOUBLE COMPLEX **for** zbbcsd

On exit, $v2t$ is premultiplied by the transpose of the right singular vector matrix common to [$b12 \ 0 \ 0 ; 0 \ -I \ 0$] and [$b22 \ 0 \ 0 ; 0 \ 0 \ I$].

$b11d$ REAL **for** sbbcsd

DOUBLE PRECISION **for** dbbcsd

COMPLEX **for** cbbcsd

DOUBLE COMPLEX **for** zbbcsd

Array, DIMENSION (q).

When ?bbcsd converges, $b11d$ contains the cosines of $\theta(1), \dots, \theta(q)$. If ?bbcsd fails to converge, $b11d$ contains the diagonal of the partially reduced top left block.

$b11e$ REAL **for** sbbcsd

DOUBLE PRECISION **for** dbbcsd

COMPLEX **for** cbbcsd

DOUBLE COMPLEX **for** zbbcsd

Array, DIMENSION ($q-1$).

When ?bbcsd converges, $b11e$ contains zeros. If ?bbcsd fails to converge, $b11e$ contains the superdiagonal of the partially reduced top left block.

$b12d$ REAL **for** sbbcsd

DOUBLE PRECISION **for** dbbcsd

COMPLEX **for** cbbcsd

DOUBLE COMPLEX **for** zbbcsd

Array, DIMENSION (q).

When ?bbcsd converges, $b12d$ contains the negative sines of $\theta(1), \dots, \theta(q)$. If ?bbcsd fails to converge, $b12d$ contains the diagonal of the partially reduced top right block.

$b12e$ REAL **for** sbbcsd

DOUBLE PRECISION **for** dbbcsd

COMPLEX **for** cbbcsd

DOUBLE COMPLEX **for** zbbcsd

Array, DIMENSION ($q-1$).

When `?bbcsd` converges, `b12e` contains zeros. If `?bbcsd` fails to converge, `b11e` contains the superdiagonal of the partially reduced top right block.

<code>info</code>	INTEGER. = 0: successful exit < 0: if <code>info = -i</code> , the i -th argument has an illegal value > 0: if <code>?bbcsd</code> did not converge, <code>info</code> specifies the number of nonzero entries in <code>phi</code> , and <code>b11d</code> , <code>b11e</code> , etc. and contains the partially reduced matrix.
-------------------	---

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `?bbcsd` interface are as follows:

<code>theta</code>	Holds the vector of length q .
<code>phi</code>	Holds the vector of length $q-1$.
<code>u1</code>	Holds the matrix of size (p,p) .
<code>u2</code>	Holds the matrix of size $(m-p,m-p)$.
<code>v1t</code>	Holds the matrix of size (q,q) .
<code>v2t</code>	Holds the matrix of size $(m-q,m-q)$.
<code>b11d</code>	Holds the vector of length q .
<code>b11e</code>	Holds the vector of length $q-1$.
<code>b12d</code>	Holds the vector of length q .
<code>b12e</code>	Holds the vector of length $q-1$.
<code>b21d</code>	Holds the vector of length q .
<code>b21e</code>	Holds the vector of length $q-1$.
<code>b22d</code>	Holds the vector of length q .
<code>b22e</code>	Holds the vector of length $q-1$.
<code>jobsu1</code>	Indicates whether u_1 is computed. Must be 'Y' or 'O'.
<code>jobsu2</code>	Indicates whether u_2 is computed. Must be 'Y' or 'O'.
<code>jobv1t</code>	Indicates whether v_1^t is computed. Must be 'Y' or 'O'.
<code>jobv2t</code>	Indicates whether v_2^t is computed. Must be 'Y' or 'O'.
<code>trans</code>	Must be 'N' or 'T'.

See Also

[?orcsd/?uncsd](#)
[xerbla](#)

?orbdb/?unbdb

Simultaneously bidiagonalizes the blocks of a partitioned orthogonal/unitary matrix.

Syntax**FORTRAN 77:**

```
call sorbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
call dorbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
call cunbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
call zunbdb( trans, signs, m, p, q, x11, ldx11, x12, ldx12, x21, ldx21, x22, ldx22,
theta, phi, taup1, taup2, tauq1, tauq2, work, lwork, info )
```

Fortran 95:

```
call orbdb( x11,x12,x21,x22,theta,phi,taup1,taup2,tauq1,tauq2[,trans][,signs][,info] )
call unbdb( x11,x12,x21,x22,theta,phi,taup1,taup2,tauq1,tauq2[,trans][,signs][,info] )
```

C:

```
lapack_int LAPACKE_sorbdb( int matrix_layout, char trans, char signs, lapack_int m,
lapack_int p, lapack_int q, float* x11, lapack_int ldx11, float* x12, lapack_int
ldx12, float* x21, lapack_int ldx21, float* x22, lapack_int ldx22, float* theta,
float* phi, float* taup1, float* taup2, float* tauq1, float* tauq2 );
lapack_int LAPACKE_dorbdb( int matrix_layout, char trans, char signs, lapack_int m,
lapack_int p, lapack_int q, double* x11, lapack_int ldx11, double* x12, lapack_int
ldx12, double* x21, lapack_int ldx21, double* x22, lapack_int ldx22, double* theta,
double* phi, double* taup1, double* taup2, double* tauq1, double* tauq );
lapack_int LAPACKE_cunbdb( int matrix_layout, char trans, char signs, lapack_int m,
lapack_int p, lapack_int q, lapack_complex_float* x11, lapack_int ldx11,
lapack_complex_float* x12, lapack_int ldx12, lapack_complex_float* x21, lapack_int
ldx21, lapack_complex_float* x22, lapack_int ldx22, float* theta, float* phi,
lapack_complex_float* taup1, lapack_complex_float* taup2, lapack_complex_float* tauq1,
lapack_complex_float* tauq2 );
lapack_int LAPACKE_zunbdb( int matrix_layout, char trans, char signs, lapack_int m,
lapack_int p, lapack_int q, lapack_complex_double* x11, lapack_int ldx11,
lapack_complex_double* x12, lapack_int ldx12, lapack_complex_double* x21, lapack_int
ldx21, lapack_complex_double* x22, lapack_int ldx22, double* theta, double* phi,
lapack_complex_double* taup1, lapack_complex_double* taup2, lapack_complex_double*
tauq1, lapack_complex_double* tauq2 );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routines `?orbdb/?unbdb` simultaneously bidiagonalizes the blocks of an m -by- m partitioned orthogonal matrix X :

$$X = \begin{pmatrix} x_{11} & | & x_{12} \\ x_{21} & | & x_{22} \end{pmatrix} = \begin{pmatrix} p_1 & | & \\ & | & p_2 \end{pmatrix} \begin{pmatrix} b_{11}|b_{12} & 0 & 0 \\ 0 | 0 & -I & 0 \\ b_{21}|b_{22} & 0 & 0 \\ 0 | 0 & 0 & I \end{pmatrix} \begin{pmatrix} q_1 & | & \\ & | & q_2 \end{pmatrix}^T$$

or unitary matrix:

$$X = \begin{pmatrix} x_{11} & | & x_{12} \\ x_{21} & | & x_{22} \end{pmatrix} = \begin{pmatrix} p_1 & | & \\ & | & p_2 \end{pmatrix} \begin{pmatrix} b_{11}|b_{12} & 0 & 0 \\ 0 | 0 & -I & 0 \\ b_{21}|b_{22} & 0 & 0 \\ 0 | 0 & 0 & I \end{pmatrix} \begin{pmatrix} q_1 & | & \\ & | & q_2 \end{pmatrix}^H$$

x_{11} is p -by- q . q must not be larger than p , $m-p$, or $m-q$. Otherwise, x must be transposed and/or permuted in constant time using the `trans` and `signs` options. See `?orcisd/?uncsd` for details.

The orthogonal/unitary matrices p_1 , p_2 , q_1 , and q_2 are p -by- p , $(m-p)$ -by- $(m-p)$, q -by- q , $(m-q)$ -by- $(m-q)$, respectively. They are represented implicitly by Householder vectors.

The bidiagonal matrices b_{11} , b_{12} , b_{21} , and b_{22} are q -by- q bidiagonal matrices represented implicitly by angles `theta(1), ..., theta(q)` and `phi(1), ..., phi(q-1)`. b_{11} and b_{12} are upper bidiagonal, while b_{21} and b_{22} are lower bidiagonal. Every entry in each bidiagonal band is a product of a sine or cosine of `theta` with a sine or cosine of `phi`. See [Sutton09] or description of `?orcisd/?uncsd` for details.

p_1 , p_2 , q_1 , and q_2 are represented as products of elementary reflectors. See description of `?orcisd/?uncsd` for details on generating p_1 , p_2 , q_1 , and q_2 using `?orgqr` and `?orglq`.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>trans</code>	CHARACTER	
= 'T':		$x, u_1, u_2, v_1^t, v_2^t$ are stored in row-major order.
otherwise		$x, u_1, u_2, v_1^t, v_2^t$ are stored in column-major order.
<code>signs</code>	CHARACTER	
= 'O':		The lower-left block is made nonpositive (the "other" convention).
otherwise		The upper-right block is made nonpositive (the "default" convention).
m		INTEGER. The number of rows and columns of the matrix X .

<i>p</i>	INTEGER. The number of rows in x_{11} and x_{12} . $0 \leq p \leq m$.
<i>q</i>	INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq \min(p, m-p, m-q)$.
<i>x₁₁</i>	<p>REAL for sorbdb DOUBLE PRECISION for dorbdb COMPLEX for cunbdb DOUBLE COMPLEX for zunbdb Array, DIMENSION (<i>ldx₁₁,q</i>).</p> <p>On entry, the top-left block of the orthogonal/unitary matrix to be reduced.</p>
<i>ldx₁₁</i>	INTEGER. The leading dimension of the array X_{11} . If <i>trans</i> = 'T', <i>ldx₁₁</i> $\geq p$. Otherwise, <i>ldx₁₁</i> $\geq q$.
<i>x₁₂</i>	<p>REAL for sorbdb DOUBLE PRECISION for dorbdb COMPLEX for cunbdb DOUBLE COMPLEX for zunbdb Array, DIMENSION (<i>ldx₁₂,m-q</i>).</p> <p>On entry, the top-right block of the orthogonal/unitary matrix to be reduced.</p>
<i>ldx₁₂</i>	INTEGER. The leading dimension of the array X_{12} . If <i>trans</i> = 'N', <i>ldx₁₂</i> $\geq p$. Otherwise, <i>ldx₁₂</i> $\geq m-q$.
<i>x₂₁</i>	<p>REAL for sorbdb DOUBLE PRECISION for dorbdb COMPLEX for cunbdb DOUBLE COMPLEX for zunbdb Array, DIMENSION (<i>ldx₂₁,q</i>).</p> <p>On entry, the bottom-left block of the orthogonal/unitary matrix to be reduced.</p>
<i>ldx₂₁</i>	INTEGER. The leading dimension of the array X_{21} . If <i>trans</i> = 'N', <i>ldx₂₁</i> $\geq m-p$. Otherwise, <i>ldx₂₁</i> $\geq q$.
<i>x₂₂</i>	<p>REAL for sorbdb DOUBLE PRECISION for dorbdb COMPLEX for cunbdb DOUBLE COMPLEX for zunbdb Array, DIMENSION (<i>ldx₂₂,m-q</i>).</p> <p>On entry, the bottom-right block of the orthogonal/unitary matrix to be reduced.</p>

<i>lidx22</i>	INTEGER. The leading dimension of the array X_{21} . If $\text{trans} = \text{'N'}$, $\text{lidx22} \geq m-p$. Otherwise, $\text{lidx22} \geq m-q$.
<i>work</i>	<p>REAL for <code>sorbdb</code></p> <p>DOUBLE PRECISION for <code>dorbdb</code></p> <p>COMPLEX for <code>cunbdb</code></p> <p>DOUBLE COMPLEX for <code>zunbdb</code></p> <p>Workspace array, DIMENSION (<i>lwork</i>).</p>
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array. $\text{lwork} \geq m-q$</p> <p>If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <code>xerbla</code>.</p>

Output Parameters

<i>x11</i>	On exit, the form depends on <i>trans</i> :
	<p>If <i>trans</i>=<code>'N'</code>, the columns of <code>tril(x11)</code> specify reflectors for p_1, the rows of <code>triu(x11, 1)</code> specify reflectors for q_1</p> <p>otherwise the rows of <code>triu(x11)</code> specify reflectors for p_1, the columns of <code>tril(x11, -1)</code> specify reflectors for q_1</p>
<i>x12</i>	On exit, the form depends on <i>trans</i> :
	<p>If <i>trans</i>=<code>'N'</code>, the columns of <code>triu(x12)</code> specify the first p reflectors for q_2</p> <p>otherwise the columns of <code>tril(x12)</code> specify the first p reflectors for q_2</p>
<i>x21</i>	On exit, the form depends on <i>trans</i> :
	<p>If <i>trans</i>=<code>'N'</code>, the columns of <code>tril(x21)</code> specify the reflectors for p_2</p> <p>otherwise the columns of <code>triu(x21)</code> specify the reflectors for p_2</p>
<i>x22</i>	On exit, the form depends on <i>trans</i> :
	<p>If <i>trans</i>=<code>'N'</code>, the rows of <code>triu(x22(q+1:m-p, p+1:m-q))</code> specify the last $m-p-q$ reflectors for q_2</p> <p>otherwise the columns of <code>tril(x22(p+1:m-q, q+1:m-p))</code> specify the last $m-p-q$ reflectors for p_2</p>
<i>theta</i>	<p>REAL for <code>sorbdb</code></p> <p>DOUBLE PRECISION for <code>dorbdb</code></p> <p>COMPLEX for <code>cunbdb</code></p> <p>DOUBLE COMPLEX for <code>zunbdb</code></p>

Array, DIMENSION (q). The entries of bidiagonal blocks b_{11} , b_{12} , b_{21} , and b_{22} can be computed from the angles *theta* and *phi*. See the Description section for details.

phi

REAL for sorbdb
 DOUBLE PRECISION for dorbdb
 COMPLEX for cunbdb
 DOUBLE COMPLEX for zunbdb

Array, DIMENSION (q-1). The entries of bidiagonal blocks b_{11} , b_{12} , b_{21} , and b_{22} can be computed from the angles *theta* and *phi*. See the Description section for details.

taup1

REAL for sorbdb
 DOUBLE PRECISION for dorbdb
 COMPLEX for cunbdb
 DOUBLE COMPLEX for zunbdb

Array, DIMENSION (p).
 Scalar factors of the elementary reflectors that define p_1 .

taup2

REAL for sorbdb
 DOUBLE PRECISION for dorbdb
 COMPLEX for cunbdb
 DOUBLE COMPLEX for zunbdb

Array, DIMENSION (m-p).
 Scalar factors of the elementary reflectors that define p_2 .

tauq1

REAL for sorbdb
 DOUBLE PRECISION for dorbdb
 COMPLEX for cunbdb
 DOUBLE COMPLEX for zunbdb

Array, DIMENSION (q).
 Scalar factors of the elementary reflectors that define q_1 .

tauq2

REAL for sorbdb
 DOUBLE PRECISION for dorbdb
 COMPLEX for cunbdb
 DOUBLE COMPLEX for zunbdb

Array, DIMENSION (m-q).
 Scalar factors of the elementary reflectors that define q_2 .

info

INTEGER.
 = 0: successful exit
 < 0: if *info* = $-i$, the *i*-th argument has an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `?orbdb/?unbdb` interface are as follows:

<code>x11</code>	Holds the block of matrix X of size (p, q) .
<code>x12</code>	Holds the block of matrix X of size $(p, m-q)$.
<code>x21</code>	Holds the block of matrix X of size $(m-p, q)$.
<code>x22</code>	Holds the block of matrix X of size $(m-p, m-q)$.
<code>theta</code>	Holds the vector of length q .
<code>phi</code>	Holds the vector of length $q-1$.
<code>taup1</code>	Holds the vector of length p .
<code>taup2</code>	Holds the vector of length $m-p$.
<code>tauq1</code>	Holds the vector of length q .
<code>taupq2</code>	Holds the vector of length $m-q$.
<code>trans</code>	Must be ' <code>N</code> ' or ' <code>T</code> '.
<code>signs</code>	Must be ' <code>O</code> ' or ' <code>D</code> '.

See Also

[?orcisd/?uncsd](#)
[?orgqr](#)
[?ungqr](#)
[?orglq](#)
[?unglq](#)
[xerbla](#)

Driver Routines

Each of the LAPACK driver routines solves a complete problem. To arrive at the solution, driver routines typically call a sequence of appropriate [computational routines](#).

Driver routines are described in the following sections :

[Linear Least Squares \(LLS\) Problems](#)
[Generalized LLS Problems](#)
[Symmetric Eigenproblems](#)
[Nonsymmetric Eigenproblems](#)
[Singular Value Decomposition](#)
[Cosine-Sine Decomposition](#)
[Generalized Symmetric Definite Eigenproblems](#)
[Generalized Nonsymmetric Eigenproblems](#)

Linear Least Squares (LLS) Problems

This section describes LAPACK driver routines used for solving linear least squares problems. [Table "Driver Routines for Solving LLS Problems"](#) lists all such routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving LLS Problems

Routine Name	Operation performed
gels	Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.
gelsy	Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.
gelss	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.
gelsd	Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

?gels

Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.

Syntax

FORTRAN 77:

```
call sgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call dgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call cgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
call zgels(trans, m, n, nrhs, a, lda, b, ldb, work, lwork, info)
```

Fortran 95:

```
call gels(a, b [,trans] [,info])
```

C:

```
lapack_int LAPACKE_<?>gels( int matrix_layout, char trans, lapack_int m, lapack_int n,
lapack_int nrhs, <datatype>* a, lapack_int lda, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves overdetermined or underdetermined real/ complex linear systems involving an m -by- n matrix A , or its transpose/ conjugate-transpose, using a QR or LQ factorization of A . It is assumed that A has full rank.

The following options are provided:

1. If $trans = 'N'$ and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$\text{minimize } ||b - A^*x||_2$

-
2. If $\text{trans} = \text{'N'}$ and $m < n$: find the minimum norm solution of an underdetermined system $A^*X = B$.
 3. If $\text{trans} = \text{'T'}$ or 'C' and $m \geq n$: find the minimum norm solution of an undetermined system $A_H^*X = B$.
 4. If $\text{trans} = \text{'T'}$ or 'C' and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

$\text{minimize } ||b - A^H X||_2$

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m -by- n_{rhs} right hand side matrix B and the n -by- n_{rh} solution matrix X .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

$trans$	CHARACTER*1. Must be ' N ', ' T ', or ' C '. If $trans = \text{'N'}$, the linear system involves matrix A ; If $trans = \text{'T'}$, the linear system involves the transposed matrix A^T (for real flavors only); If $trans = \text{'C'}$, the linear system involves the conjugate-transposed matrix A^H (for complex flavors only).
m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns of the matrix A ($n \geq 0$).
n_{rhs}	INTEGER. The number of right-hand sides; the number of columns in B ($n_{rhs} \geq 0$).
$a, b, work$	REAL for sgels DOUBLE PRECISION for dgels COMPLEX for cgels DOUBLE COMPLEX for zgels. Arrays: $a(lda,*)$ contains the m -by- n matrix A . The second dimension of a must be at least $\max(1, n)$. $b(lsb,*)$ contains the matrix B of right hand side vectors, stored columnwise; B is m -by- n_{rhs} if $trans = \text{'N'}$, or n -by- n_{rhs} if $trans = \text{'T'}$ or ' C '. The second dimension of b must be at least $\max(1, n_{rhs})$. $work$ is a workspace array, its dimension $\max(1, lwork)$.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
lsb	INTEGER. The leading dimension of b ; must be at least $\max(1, m, n)$.

lwork INTEGER. The size of the *work* array; must be at least $\min(m, n) + \max(1, m, n, nrhs)$.
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.
 See *Application Notes* for the suggested value of *lwork*.

Output Parameters

a On exit, overwritten by the factorization data as follows:
 if $m \geq n$, array *a* contains the details of the QR factorization of the matrix *A* as returned by ?geqrf;
 if $m < n$, array *a* contains the details of the LQ factorization of the matrix *A* as returned by ?gelqf.

b If *info* = 0, *b* overwritten by the solution vectors, stored columnwise:
 if *trans* = 'N' and $m \geq n$, rows 1 to *n* of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements *n+1* to *m* in that column;
 if *trans* = 'N' and $m < n$, rows 1 to *n* of *b* contain the minimum norm solution vectors;
 if *trans* = 'T' or 'C' and $m \geq n$, rows 1 to *m* of *b* contain the minimum norm solution vectors; if *trans* = 'T' or 'C' and $m < n$, rows 1 to *m* of *b* contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements *m+1* to *n* in that column.

work(1) If *info* = 0, on exit *work(1)* contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.
 If *info* = *i*, the *i*-th diagonal element of the triangular factor of *A* is zero, so that *A* does not have full rank; the least squares solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *gels* interface are the following:

a Holds the matrix *A* of size (m,n) .
b Holds the matrix of size $\max(m,n)$ -by-*nrhs*.
 If *trans* = 'N', then, on entry, the size of *b* is *m*-by-*nrhs*,

If $\text{trans} = \text{'T'}$, then, on entry, the size of b is n -by- $nrhs$,
 trans Must be ' N ' or ' T '. The default value is ' N '.

Application Notes

For better performance, try using $lwork = \min(m, n) + \max(1, m, n, nrhs) * blocksize$, where $blocksize$ is a machine-dependent value (typically, 16 to 64) required for optimum performance of the *blocked algorithm*.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelsy

Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A .

Syntax

FORTRAN 77:

```
call sgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, info)
call dgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, info)
call cgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, rwork, info)
call zgelsy(m, n, nrhs, a, lda, b, ldb, jpvt, rcond, rank, work, lwork, rwork, info)
```

Fortran 95:

```
call gelsy(a, b [,rank] [,jpvt] [,rcond] [,info])
```

C:

```
lapack_int LAPACKE_sgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, float* a, lapack_int lda, float* b, lapack_int ldb, lapack_int* jpvt, float
rcond, lapack_int* rank );
```

```
lapack_int LAPACKE_dgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, double* a, lapack_int lda, double* b, lapack_int ldb, lapack_int* jpvt, double
rcond, lapack_int* rank );
```

```
lapack_int LAPACKE_cgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int
ldb, lapack_int* jpvt, float rcond, lapack_int* rank );
```

```
lapack_int LAPACKE_zgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int
ldb, lapack_int* jpvt, double rcond, lapack_int* rank );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The ?gelsy routine computes the minimum-norm solution to a real/complex linear least squares problem:

```
minimize ||b - A*x||_2
```

using a complete orthogonal factorization of A . A is an m -by- n matrix which may be rank-deficient. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m -by- $nrhs$ right hand side matrix B and the n -by- $nrhs$ solution matrix X .

The routine first computes a QR factorization with column pivoting:

$$\dots \begin{pmatrix} \dots & \dots & \dots & \dots & \dots \end{pmatrix}$$

with R_{11} defined as the largest leading submatrix whose estimated condition number is less than $1/rcond$. The order of R_{11} , *rank*, is the effective rank of A . Then, R_{22} is considered to be negligible, and R_{12} is annihilated by orthogonal/unitary transformations from the right, arriving at the complete orthogonal factorization:

$$AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z$$

The minimum-norm solution is then

$$X = PZ^T \begin{pmatrix} T_{11}^{-1} & Q_1^T b \\ 0 & 0 \end{pmatrix} \text{ for real flavors and}$$

$$\dots \begin{pmatrix} \dots & \dots & \dots & \dots & \dots \end{pmatrix}$$

for complex flavors,

where Q_1 consists of the first *rank* columns of Q .

The ?gelsy routine is identical to the original deprecated ?gelsx routine except for the following differences:

- The call to the subroutine ?geqpf has been substituted by the call to the subroutine ?geqp3, which is a BLAS-3 version of the QR factorization with column pivoting.
- The matrix B (the right hand side) is updated with BLAS-3.
- The permutation of the matrix B (the right hand side) is faster and more simple.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns in <i>B</i> ($nrhs \geq 0$).
<i>a, b, work</i>	<p>REAL for sgelsy DOUBLE PRECISION for dgelsy COMPLEX for cgelsy DOUBLE COMPLEX for zgelsy.</p> <p>Arrays:</p> <p><i>a</i>(<i>lda,*</i>) contains the <i>m</i>-by-<i>n</i> matrix <i>A</i>. The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i>(<i>ldb,*</i>) contains the <i>m</i>-by-<i>nrhs</i> right hand side matrix <i>B</i>. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.</p>
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; must be at least $\max(1, m, n)$.
<i>jpvt</i>	<p>INTEGER. Array, DIMENSION at least $\max(1, n)$. On entry, if $jpvt(i) \neq 0$, the <i>i</i>-th column of <i>A</i> is permuted to the front of <i>AP</i>, otherwise the <i>i</i>-th column of <i>A</i> is a free column.</p>
<i>rcond</i>	<p>REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. <i>rcond</i> is used to determine the effective rank of <i>A</i>, which is defined as the order of the largest leading triangular submatrix <i>R</i>₁₁ in the QR factorization with pivoting of <i>A</i>, whose estimated condition number < 1/<i>rcond</i>.</p>
<i>lwork</i>	<p>INTEGER. The size of the <i>work</i> array. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i>. See <i>Application Notes</i> for the suggested value of <i>lwork</i>.</p>
<i>rwork</i>	REAL for cgelsy DOUBLE PRECISION for zgelsy. Workspace array, DIMENSION at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

<i>a</i>	On exit, overwritten by the details of the complete orthogonal factorization of A .
<i>b</i>	Overwritten by the n -by- $nrhs$ solution matrix X .
<i>jpvt</i>	On exit, if $jpvt(i) = k$, then the i -th column of AP was the k -th column of A .
<i>rank</i>	INTEGER. The effective rank of A , that is, the order of the submatrix R_{11} . This is the same as the order of the submatrix T_{11} in the complete orthogonal factorization of A .
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gelsy` interface are the following:

<i>a</i>	Holds the matrix A of size (m,n) .
<i>b</i>	Holds the matrix of size $\max(m,n)$ -by- $nrhs$. On entry, contains the m -by- $nrhs$ right hand side matrix B , On exit, overwritten by the n -by- $nrhs$ solution matrix X .
<i>jpvt</i>	Holds the vector of length n . Default value for this element is $jpvt(i) = 0$.
<i>rcond</i>	Default value for this element is $rcond = 100*\text{EPSILON}(1.0_WP)$.

Application Notes

For real flavors:

The unblocked strategy requires that:

$$lwork \geq \max(mn+3n+1, 2*mn + nrhs),$$

where $mn = \min(m, n)$.

The block algorithm requires that:

$$lwork \geq \max(mn+2n+nb*(n+1), 2*mn+nb*nrhs),$$

where nb is an upper bound on the blocksize returned by `ilaenv` for the routines `sgeqp3/dgeqp3, stzrzf/dtzrzf, stzrqf/dtzrqf, sormqr/dormqr, and sormrz/dormrz`.

For complex flavors:

The unblocked strategy requires that:

$$lwork \geq mn + \max(2*mn, n+1, mn + nrhs),$$

where $mn = \min(m, n)$.

The block algorithm requires that:

$$lwork < mn + \max(2*mn, nb*(n+1), mn+mn*nb, mn+ nb*nrhs),$$

where nb is an upper bound on the blocksize returned by [ilaenv](#) for the routines cgeqp3/zgeqp3, ctzrzf/ztzrjf, ctzrqf/ztzrqf, cunmqr/zunmqr, and cunmrz/zunmrz.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelss

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.

Syntax

FORTRAN 77:

```
call sgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, info)
call dgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, info)
call cgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, info)
call zgelss(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, info)
```

Fortran 95:

```
call gelss(a, b [,rank] [,s] [,rcond] [,info])
```

C:

```
lapack_int LAPACKE_sgelss( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, float* a, lapack_int lda, float* b, lapack_int ldb, float* s, float rcond,
lapack_int* rank );
lapack_int LAPACKE_dgelss( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, double* a, lapack_int lda, double* b, lapack_int ldb, double* s, double rcond,
lapack_int* rank );
lapack_int LAPACKE_cgelss( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int
ldb, float* s, float rcond, lapack_int* rank );
lapack_int LAPACKE_zgelss( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int
ldb, double* s, double rcond, lapack_int* rank );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the minimum norm solution to a real linear least squares problem:

```
minimize ||b - A*x||_2
```

using the singular value decomposition (SVD) of A . A is an m -by- n matrix which may be rank-deficient. Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m -by- $nrhs$ right hand side matrix B and the n -by- $nrhs$ solution matrix X . The effective rank of A is determined by treating as zero those singular values which are less than $rcond$ times the largest singular value.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m	INTEGER. The number of rows of the matrix A ($m \geq 0$).
n	INTEGER. The number of columns of the matrix A ($n \geq 0$).
$nrhs$	INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
$a, b, work$	<p>REAL for sgels DOUBLE PRECISION for dgels COMPLEX for cgels DOUBLE COMPLEX for zgels.</p> <p>Arrays:</p> <p>$a(lda,*)$ contains the m-by-n matrix A. The second dimension of a must be at least $\max(1, n)$.</p> <p>$b ldb,*)$ contains the m-by-$nrhs$ right hand side matrix B. The second dimension of b must be at least $\max(1, nrhs)$.</p> <p>$work$ is a workspace array, its dimension $\max(1, lwork)$.</p>
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
ldb	INTEGER. The leading dimension of b ; must be at least $\max(1, m, n)$.
$rcond$	<p>REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.</p> <p>$rcond$ is used to determine the effective rank of A. Singular values $s(i) \leq rcond * s(1)$ are treated as zero. If $rcond < 0$, machine precision is used instead.</p>
$lwork$	INTEGER. The size of the $work$ array; $lwork \geq 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

See *Application Notes* for the suggested value of $lwork$.

$rwork$

REAL for `cgelss`

DOUBLE PRECISION for `zgelss`.

Workspace array used in complex flavors only. DIMENSION at least $\max(1, 5 * \min(m, n))$.

Output Parameters

a

On exit, the first $\min(m, n)$ rows of A are overwritten with its right singular vectors, stored row-wise.

b

Overwritten by the n -by- $nrhs$ solution matrix X .

If $m \geq n$ and $rank = n$, the residual sum-of-squares for the solution in the i -th column is given by the sum of squares of modulus of elements $n+1:m$ in that column.

s

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, \min(m, n))$. The singular values of A in decreasing order. The condition number of A in the 2-norm is

$$k_2(A) = s(1) / s(\min(m, n)).$$

$rank$

INTEGER. The effective rank of A , that is, the number of singular values which are greater than $rcond * s(1)$.

$work(1)$

If $info = 0$, on exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance. Use this $lwork$ for subsequent runs.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then the algorithm for computing the SVD failed to converge; i indicates the number of off-diagonal elements of an intermediate bidiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gelss` interface are the following:

a

Holds the matrix A of size (m,n) .

<i>b</i>	Holds the matrix of size $\max(m,n)$ -by- $nrhs$. On entry, contains the m -by- $nrhs$ right hand side matrix B , On exit, overwritten by the n -by- $nrhs$ solution matrix X .
<i>s</i>	Holds the vector of length $\min(m,n)$.
<i>rcond</i>	Default value for this element is $rcond = 100 * \text{EPSILON}(1.0_WP)$.

Application Notes

For real flavors:

$$lwork \geq 3 * \min(m, n) + \max(2 * \min(m, n), \max(m, n), nrhs)$$

For complex flavors:

$$lwork \geq 2 * \min(m, n) + \max(m, n, nrhs)$$

For good performance, *lwork* should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gelsd

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

Syntax

FORTRAN 77:

```
call sgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork, info)
call dgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, iwork, info)
call cgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, iwork,
info)
call zgelsd(m, n, nrhs, a, lda, b, ldb, s, rcond, rank, work, lwork, rwork, iwork,
info)
```

Fortran 95:

```
call gelsd(a, b [,rank] [,s] [,rcond] [,info])
```

C:

```
lapack_int LAPACKE_sgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, float* a, lapack_int lda, float* b, lapack_int ldb, float* s, float rcond,
lapack_int* rank );
lapack_int LAPACKE_dgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, double* a, lapack_int lda, double* b, lapack_int ldb, double* s, double rcond,
lapack_int* rank );
```

```
lapack_int LAPACKE_cgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int
ldb, float* s, float rcond, lapack_int* rank );
```

```
lapack_int LAPACKE_zgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int
nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int
ldb, double* s, double rcond, lapack_int* rank );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the minimum-norm solution to a real linear least squares problem:

$\text{minimize } \|b - Ax\|_2$

using the singular value decomposition (SVD) of A . A is an m -by- n matrix which may be rank-deficient.

Several right hand side vectors b and solution vectors x can be handled in a single call; they are stored as the columns of the m -by- $nrhs$ right hand side matrix B and the n -by- $nrhs$ solution matrix X .

The problem is solved in three steps:

1. Reduce the coefficient matrix A to bidiagonal form with Householder transformations, reducing the original problem into a "bidiagonal least squares problem" (BLS).
2. Solve the BLS using a divide and conquer approach.
3. Apply back all the Householder transformations to solve the original least squares problem.

The effective rank of A is determined by treating as zero those singular values which are less than $rcond$ times the largest singular value.

The routine uses auxiliary routines [lals0](#) and [lalsa](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m INTEGER. The number of rows of the matrix A ($m \geq 0$).

n INTEGER. The number of columns of the matrix A ($n \geq 0$).

nrhs INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).

a, b, work
REAL for sgelsd
DOUBLE PRECISION for dgelsd
COMPLEX for cgelsd
DOUBLE COMPLEX for zgelsd.

Arrays:

a(*lda,**) contains the m -by- n matrix A .

The second dimension of a must be at least $\max(1, n)$.

$b(lDb, *)$ contains the m -by- $nrhs$ right hand side matrix B .

The second dimension of b must be at least $\max(1, nrhs)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

ldb INTEGER. The leading dimension of b ; must be at least $\max(1, m, n)$.

$rcond$ REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

$rcond$ is used to determine the effective rank of A . Singular values $s(i) \leq rcond * s(1)$ are treated as zero. If $rcond \leq 0$, machine precision is used instead.

$lwork$ INTEGER. The size of the $work$ array; $lwork \geq 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the array $work$ and the minimum sizes of the arrays $rwork$ and $iwork$, and returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ is issued by [xerbla](#).

See [Application Notes](#) for the suggested value of $lwork$.

$iwork$ INTEGER. Workspace array. See [Application Notes](#) for the suggested dimension of $iwork$.

$rwork$ REAL for `cgelsd`

DOUBLE PRECISION for `zgelsd`.

Workspace array, used in complex flavors only. See [Application Notes](#) for the suggested dimension of $rwork$.

Output Parameters

a On exit, A has been overwritten.

b Overwritten by the n -by- $nrhs$ solution matrix X .

If $m \geq n$ and $rank = n$, the residual sum-of-squares for the solution in the i -th column is given by the sum of squares of modulus of elements $n+1:m$ in that column.

s REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, \min(m, n))$. The singular values of A in decreasing order. The condition number of A in the 2-norm is

$$\kappa^2(A) = s(1) / s(\min(m, n)).$$

$rank$ INTEGER. The effective rank of A , that is, the number of singular values which are greater than $rcond * s(1)$.

<i>work</i> (1)	If $info = 0$, on exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance. Use this <i>lwork</i> for subsequent runs.
<i>rwork</i> (1)	If $info = 0$, on exit, <i>rwork</i> (1) returns the minimum size of the workspace array <i>iwork</i> required for optimum performance.
<i>iwork</i> (1)	If $info = 0$, on exit, <i>iwork</i> (1) returns the minimum size of the workspace array <i>iwork</i> required for optimum performance.
<i>info</i>	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value. If $info = i$, then the algorithm for computing the SVD failed to converge; i indicates the number of off-diagonal elements of an intermediate bidiagonal form that did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gelsd` interface are the following:

<i>a</i>	Holds the matrix A of size (m,n) .
<i>b</i>	Holds the matrix of size $\max(m,n)$ -by- $nrhs$. On entry, contains the m -by- $nrhs$ right hand side matrix B , On exit, overwritten by the n -by- $nrhs$ solution matrix X .
<i>s</i>	Holds the vector of length $\min(m,n)$.
<i>rcond</i>	Default value for this element is $rcond = 100 * \text{EPSILON}(1.0_WP)$.

Application Notes

The divide and conquer algorithm makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

The exact minimum amount of workspace needed depends on m , n and $nrhs$. The size *lwork* of the workspace array *work* must be as given below.

For real flavors:

If $m \geq n$,

$$lwork \geq 12n + 2n*smlsiz + 8n*nlvl + n*nrhs + (smlsiz+1)^2;$$

If $m < n$,

$$lwork \geq 12m + 2m*smlsiz + 8m*nlvl + m*nrhs + (smlsiz+1)^2;$$

For complex flavors:

If $m \geq n$,

$$lwork \geq 2n + n*nrhs;$$

If $m < n$,

$$lwork \geq 2m + m*nrhs;$$

where *smlsiz* is returned by *ilaenv* and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and

```
nlvl = INT( log2( min( m, n )/(smlsiz+1) ) ) + 1.
```

For good performance, *lwork* should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The dimension of the workspace array *iwork* must be at least

$$3*\min(m, n)*nlvl + 11*\min(m, n).$$

The dimension of the workspace array *iwork* (for complex flavors) must be at least *max*(1, *lrwork*).

$$lrwork \geq 10n + 2n*smlsiz + 8n*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2 \text{ if } m \geq n, \text{ and}$$

$$lrwork \geq 10m + 2m*smlsiz + 8m*nlvl + 3*smlsiz*nrhs + (smlsiz+1)^2 \text{ if } m < n.$$

Generalized LLS Problems

This section describes LAPACK driver routines used for solving generalized linear least squares problems. Table "Driver Routines for Solving Generalized LLS Problems" lists all such routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Generalized LLS Problems

Routine Name	Operation performed
gglse	Solves the linear equality-constrained least squares problem using a generalized RQ factorization.
ggglm	Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

?gglse

Solves the linear equality-constrained least squares problem using a generalized RQ factorization.

Syntax

FORTRAN 77:

```
call sgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call dgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call cgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
call zgglse(m, n, p, a, lda, b, ldb, c, d, x, work, lwork, info)
```

Fortran 95:

```
call gglse(a, b, c, d, x [,info])
```

C:

```
lapack_int LAPACKE_<?>gglse( int matrix_layout, lapack_int m, lapack_int n, lapack_int p,
<datatype>* a, lapack_int lda, <datatype>* b, lapack_int ldb, <datatype>* c,
<datatype>* d, <datatype>* x );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine solves the linear equality-constrained least squares (LSE) problem:

$\text{minimize } ||c - A^*x||^2 \text{ subject to } B^*x = d$

where A is an m -by- n matrix, B is a p -by- n matrix, c is a given d is a given p -vector. It is assumed that $p \leq n \leq m+p$, and

$$(B^*B)^{-1} B^*A = I_m \quad \text{and} \quad B^*B \text{ is nonsingular}$$

These conditions ensure that the LSE problem has a unique solution, which is obtained using a generalized RQ factorization of the matrices (B, A) given by

$B = (0 \ R)^* Q, \quad A = Z^* T^* Q$

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m INTEGER. The number of rows of the matrix A ($m \geq 0$).

n INTEGER. The number of columns of the matrices A and B ($n \geq 0$).

p INTEGER. The number of rows of the matrix B
($0 \leq p \leq n \leq m+p$).

a, b, c, d, work
REAL for sgglse
DOUBLE PRECISION for dgglse
COMPLEX for cgglse
DOUBLE COMPLEX for zgglse.

Arrays:

a(*lda,**) contains the m -by- n matrix A .

The second dimension of *a* must be at least $\max(1, n)$.

b(*ldb,**) contains the p -by- n matrix B .

The second dimension of *b* must be at least $\max(1, n)$.

c(*), dimension at least max(1, *m*), contains the right hand side vector for the least squares part of the LSE problem.

d(*), dimension at least max(1, *p*), contains the right hand side vector for the constrained equation.

work is a workspace array, its dimension max(1, *lwork*).

lda

INTEGER. The leading dimension of *a*; at least max(1, *m*).

ldb

INTEGER. The leading dimension of *b*; at least max(1, *p*).

lwork

INTEGER. The size of the *work* array;

lwork $\geq \max(1, m+n+p)$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

See *Application Notes* for the suggested value of *lwork*.

Output Parameters

x

REAL for sgglse

b

On exit, the upper triangle of the subarray *b*(1:*p*, *n*-*p*+1:*n*) contains the *p*-by-*p* upper triangular matrix *R*.

d

On exit, *d* is destroyed.

c

On exit, the residual sum-of-squares for the solution is given by the sum of squares of elements *n*-*p*+1 to *m* of vector *c*.

work(1)

If *info* = 0, on exit, *work*(1) contains the minimum value of *lwork* required for optimum performance. Use this *lwork* for subsequent runs.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = 1, the upper triangular factor *R* associated with *B* in the generalized RQ factorization of the pair (*B*, *A*) is singular, so that $\text{rank}(B) < P$; the least squares solution could not be computed.

If *info* = 2, the (*n*-*p*)-by- (*n*-*p*) part of the upper trapezoidal factor *T* associated with *A* in the generalized RQ factorization of the pair (*B*, *A*) is singular, so that

$$\text{rank} \begin{pmatrix} A \\ B \end{pmatrix} < n$$

; the least squares solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggglm` interface are the following:

- `a` Holds the matrix A of size (m,n) .
- `b` Holds the matrix B of size (p,n) .
- `c` Holds the vector of length (m) .
- `d` Holds the vector of length (p) .
- `x` Holds the vector of length n .

Application Notes

For optimum performance, use

$$lwork \geq p + \min(m, n) + \max(m, n) * nb,$$

where nb is an upper bound for the optimal blocksizes for `?geqrfa`, `?gerqf`, `?ormqr`/`?unmqr` and `?ormrq`/`?unmrg`.

You may set $lwork$ to -1. The routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?ggglm

Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

Syntax

FORTRAN 77:

```
call sgglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call dgglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call cgglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
call zgglm(n, m, p, a, lda, b, ldb, d, x, y, work, lwork, info)
```

Fortran 95:

```
call ggglm(a, b, d, x, y [,info])
```

C:

```
lapack_int LAPACKE_<?>ggglm( int matrix_layout, lapack_int n, lapack_int m, lapack_int p, <datatype>* a, lapack_int lda, <datatype>* b, lapack_int ldb, <datatype>* d, <datatype>* x, <datatype>* y );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine solves a general Gauss-Markov linear model (GLM) problem:

$\text{minimize}_x \|y\|_2 \text{ subject to } d = A^*x + B^*y$

where A is an n -by- m matrix, B is an n -by- p matrix, and d is a given n -vector. It is assumed that $m \leq n \leq m+p$, and $\text{rank}(A) = m$ and $\text{rank}(A \ B) = n$.

Under these assumptions, the constrained equation is always consistent, and there is a unique solution x and a minimal 2-norm solution y , which is obtained using a generalized QR factorization of the matrices (A, B) given by

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}; \quad B = Q^* T^* Z.$$

In particular, if matrix B is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

$\text{minimize}_x \|B^{-1}(d - A^*x)\|_2$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

n INTEGER. The number of rows of the matrices A and B ($n \geq 0$).

m INTEGER. The number of columns in A ($m \geq 0$).

p INTEGER. The number of columns in B ($p \geq n - m$).

a, b, d, work REAL for sggglm

DOUBLE PRECISION for dggglm

COMPLEX for cggglm

DOUBLE COMPLEX for zggglm.

Arrays:

a(*lda,**) contains the n -by- m matrix A .

The second dimension of *a* must be at least $\max(1, m)$.

b(*ldb,**) contains the n -by- p matrix B .

The second dimension of *b* must be at least $\max(1, p)$.

d(*), dimension at least $\max(1, n)$, contains the left hand side of the GLM equation.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

ldb INTEGER. The leading dimension of *b*; at least $\max(1, n)$.

lwork INTEGER. The size of the *work* array; *lwork* $\geq \max(1, n+m+p)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

x, y

REAL for `sggglm`

DOUBLE PRECISION for `dggglm`

COMPLEX for `cggglm`

DOUBLE COMPLEX for `zggglm`.

Arrays $x(*)$, $y(*)$. DIMENSION at least $\max(1, m)$ for x and at least $\max(1, p)$ for y .

On exit, x and y are the solutions of the GLM problem.

a

On exit, the upper triangular part of the array a contains the m -by- m upper triangular matrix R .

b

On exit, if $n \leq p$, the upper triangle of the subarray $b(1:n, p-n+1:p)$ contains the n -by- n upper triangular matrix T ; if $n > p$, the elements on and above the $(n-p)$ -th subdiagonal contain the n -by- p upper trapezoidal matrix T .

d

On exit, d is destroyed

$work(1)$

If $info = 0$, on exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = 1$, the upper triangular factor R associated with A in the generalized QR factorization of the pair (A, B) is singular, so that $\text{rank}(A) < m$; the least squares solution could not be computed.

If $info = 2$, the bottom $(n-m)$ -by- $(n-m)$ part of the upper trapezoidal factor T associated with B in the generalized QR factorization of the pair (A, B) is singular, so that $\text{rank}(A B) < n$; the least squares solution could not be computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggglm` interface are the following:

a Holds the matrix A of size (n,m) .

b Holds the matrix B of size (n,p) .

<i>d</i>	Holds the vector of length <i>n</i> .
<i>x</i>	Holds the vector of length (<i>m</i>).
<i>y</i>	Holds the vector of length (<i>p</i>).

Application Notes

For optimum performance, use

$$lwork \geq m + \min(n, p) + \max(n, p) * nb,$$

where *nb* is an upper bound for the optimal blocksizes for ?geqr, ?gerqf, ?ormqr/?unmqr and ?ormrq/?unmrq.

You may set *lwork* to -1. The routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Symmetric Eigenproblems

This section describes LAPACK driver routines used for solving symmetric eigenvalue problems. See also [computational routines](#) that can be called to solve these problems. [Table "Driver Routines for Solving Symmetric Eigenproblems"](#) lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Symmetric Eigenproblems

Routine Name	Operation performed
syev/heev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix.
syevd/heevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix using divide and conquer algorithm.
syevx/heevx	Computes selected eigenvalues and, optionally, eigenvectors of a symmetric / Hermitian matrix.
syevr/heevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix using the Relatively Robust Representations.
spev/hpev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
spevd/hpevd	Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix held in packed storage.
spevx/hpevx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.
sbev /hbev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
sbevd/hbevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian band matrix using divide and conquer algorithm.
sbevx/hbevx	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.
stev	Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.

Routine Name	Operation performed
stevd	Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.
stevx	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
stevr	Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

?syev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix.

Syntax**FORTRAN 77:**

```
call ssyev(jobz, uplo, n, a, lda, w, work, lwork, info)
call dsyev(jobz, uplo, n, a, lda, w, work, lwork, info)
```

Fortran 95:

```
call syev(a, w [,jobz] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>syev( int matrix_layout, char jobz, char uplo, lapack_int n,
<datatype>* a, lapack_int lda, <datatype>* w );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A .

Note that for most cases of real symmetric eigenvalue problems the default choice should be [syevr](#) function as its underlying algorithm is faster and uses less workspace.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>jobz</code> <code>uplo</code>	CHARACTER*1. Must be 'N' or 'V'. If <code>jobz</code> = 'N', then only eigenvalues are computed. If <code>jobz</code> = 'V', then eigenvalues and eigenvectors are computed. CHARACTER*1. Must be 'U' or 'L'. If <code>uplo</code> = 'U', <code>a</code> stores the upper triangular part of A . If <code>uplo</code> = 'L', <code>a</code> stores the lower triangular part of A .
--	--

<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a, work</i>	REAL for ssyev DOUBLE PRECISION for dsyev
	Arrays: <i>a</i> (<i>lda</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraint: $lwork \geq \max(1, 3n-1)$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .

Output Parameters

<i>a</i>	On exit, if <i>jobz</i> = 'V', then if <i>info</i> = 0, array <i>a</i> contains the orthonormal eigenvectors of the matrix <i>A</i> . If <i>jobz</i> = 'N', then on exit the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>w</i>	REAL for ssyev DOUBLE PRECISION for dsyev Array, DIMENSION at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order.
<i>work</i> (1)	On exit, if <i>lwork</i> > 0, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `syev` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>w</code>	Holds the vector of length n .
<code>job</code>	Must be ' <code>N</code> ' or ' <code>V</code> '. The default value is ' <code>N</code> '.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

For optimum performance set $lwork \geq (nb+2)*n$, where nb is the blocksize for `?sytrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run, or set $lwork = -1$.

If $lwork$ has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array on exit. Use this value (`work(1)`) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array `work`. This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1 , then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?heev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call cheev(jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
call zheev(jobz, uplo, n, a, lda, w, work, lwork, rwork, info)
```

Fortran 95:

```
call heev(a, w [,jobz] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_cheev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_float* a, lapack_int lda, float* w );
```

```
lapack_int LAPACKE_zheev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_double* a, lapack_int lda, double* w );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A .

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be [heevr](#) function as its underlying algorithm is faster and uses less workspace.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of A . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>a</i> , <i>work</i>	COMPLEX for cheev DOUBLE COMPLEX for zheev Arrays: <i>a</i> (<i>lda</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix A , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . C onstraint: <i>lwork</i> $\geq \max(1, 2n-1)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

See *Application Notes* for the suggested value of $lwork$.

$rwork$

REAL for `cheev`

DOUBLE PRECISION for `zheev`.

Workspace array, DIMENSION at least $\max(1, 3n-2)$.

Output Parameters

a

On exit, if $jobz = 'V'$, then if $info = 0$, array a contains the orthonormal eigenvectors of the matrix A .

If $jobz = 'N'$, then on exit the lower triangle

(if $uplo = 'L'$) or the upper triangle (if $uplo = 'U'$) of A , including the diagonal, is overwritten.

w

REAL for `cheev`

DOUBLE PRECISION for `zheev`

Array, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix A in ascending order.

$work(1)$

On exit, if $lwork > 0$, then $work(1)$ returns the required minimal size of $lwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `heev` interface are the following:

a Holds the matrix A of size (n, n) .

w Holds the vector of length n .

job Must be ' N ' or ' V '. The default value is ' N '.

$uplo$ Must be ' U ' or ' L '. The default value is ' U '.

Application Notes

For optimum performance use

$$lwork \geq (nb+1)*n,$$

where nb is the blocksize for `?hetrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?syevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix using divide and conquer algorithm.

Syntax

FORTRAN 77:

```
call ssyevd(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
call dsyevd(jobz, uplo, n, a, lda, w, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call syevd(a, w [,jobz] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>syevd( int matrix_layout, char jobz, char uplo, lapack_int n,
<datatype>* a, lapack_int lda, <datatype>* w );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A . In other words, it can compute the spectral factorization of A as: $A = Z^* \Lambda^* Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Note that for most cases of real symmetric eigenvalue problems the default choice should be [syevr](#) function as its underlying algorithm is faster and uses less workspace. [?syevd](#) requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> (<i>n</i> \geq 0).
<i>a</i>	REAL for ssyevd DOUBLE PRECISION for dsyevd Array, DIMENSION (<i>lda</i> , *). <i>a</i> (<i>lدا</i> , *) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lда</i>	INTEGER. The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>work</i>	REAL for ssyevd DOUBLE PRECISION for dsyevd . Workspace array, DIMENSION at least <i>lwork</i> .
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraints: if <i>n</i> \leq 1, then <i>lwork</i> \geq 1; if <i>jobz</i> = 'N' and <i>n</i> $>$ 1, then <i>lwork</i> \geq $2*n + 1$; if <i>jobz</i> = 'V' and <i>n</i> $>$ 1, then <i>lwork</i> \geq $2*n^2 + 6*n + 1$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the required sizes of the <i>work</i> and <i>iwork</i> arrays, returns these values as the first entries of the <i>work</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> or <i>iwork</i> is issued by xerbla . See <i>Application Notes</i> for details.
<i>iwork</i>	INTEGER.

Workspace array, its dimension $\max(1, liwork)$.

$liwork$

INTEGER.

The dimension of the array $iwork$.

Constraints:

if $n \leq 1$, then $liwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $liwork \geq 1$;

if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n + 3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w

REAL for `ssyevd`

DOUBLE PRECISION for `dsyevd`

Array, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix A in ascending order.
See also $info$.

a

If $jobz = 'V'$, then on exit this array is overwritten by the orthogonal matrix Z which contains the eigenvectors of A .

$work(1)$

On exit, if $lwork > 0$, then $work(1)$ returns the required minimal size of $lwork$.

$iwork(1)$

On exit, if $liwork > 0$, then $iwork(1)$ returns the required minimal size of $liwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = i$, and $jobz = 'N'$, then the algorithm failed to converge; i indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If $info = i$, and $jobz = 'V'$, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $info/(n+1)$ through $\text{mod}(info, n+1)$.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `syevd` interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>w</i>	Holds the vector of length n .
<i>jobz</i>	Must be ' N ' or ' V '. The default value is ' N '.
<i>uplo</i>	Must be ' U ' or ' L '. The default value is ' U '.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\varepsilon) * \|T\|_2$, where ε is the machine precision.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run, or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is [heevd](#)

?heevd

Computes all eigenvalues and (optionally) all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.

Syntax

FORTRAN 77:

```
call cheevd(jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork, liwork, info)
call zheevd(jobz, uplo, n, a, lda, w, work, lwork, rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call heevd(a, w [,job] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_cheevd( int matrix_layout, char jobz, char uplo, lapack_int n,
                           lapack_complex_float* a, lapack_int lda, float* w );
lapack_int LAPACKE_zheevd( int matrix_layout, char jobz, char uplo, lapack_int n,
                           lapack_complex_double* a, lapack_int lda, double* w );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A . In other words, it can compute the spectral factorization of A as: $A = Z^* \Lambda^* Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be [heevr](#) function as its underlying algorithm is faster and uses less workspace. [?heevd](#) requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be ' <i>N</i> ' or ' <i>V</i> '. If <i>jobz</i> = ' <i>N</i> ', then only eigenvalues are computed. If <i>jobz</i> = ' <i>V</i> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '. If <i>uplo</i> = ' <i>U</i> ', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = ' <i>L</i> ', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> (<i>n</i> \geq 0).
<i>a</i>	COMPLEX for cheevd DOUBLE COMPLEX for zheevd Array, DIMENSION (<i>lda</i> , *). <i>a</i> (<i>lda</i> , *) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>work</i>	COMPLEX for cheevd DOUBLE COMPLEX for zheevd . Workspace array, DIMENSION $\max(1, lwork)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraints: if <i>n</i> \leq 1, then <i>lwork</i> \geq 1; if <i>jobz</i> = ' <i>N</i> ' and <i>n</i> $>$ 1, then <i>lwork</i> \geq <i>n</i> +1; if <i>jobz</i> = ' <i>V</i> ' and <i>n</i> $>$ 1, then <i>lwork</i> \geq <i>n</i> ² +2*i <i>n</i> .

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See [Application Notes](#) for details.

$rwork$ REAL for `cheevd`

DOUBLE PRECISION for `zheevd`

Workspace array, DIMENSION at least $lrwork$.

$lrwork$ INTEGER.

The dimension of the array $rwork$. Constraints:

if $n \leq 1$, then $lrwork \geq 1$;

if $job = 'N'$ and $n > 1$, then $lrwork \geq n$;

if $job = 'V'$ and $n > 1$, then $lrwork \geq 2*n^2 + 5*n + 1$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See [Application Notes](#) for details.

$iwork$ INTEGER. Workspace array, its dimension $\max(1, liwork)$.

$liwork$ INTEGER.

The dimension of the array $iwork$. Constraints: if $n \leq 1$, then $liwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $liwork \geq 1$;

if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See [Application Notes](#) for details.

Output Parameters

w REAL for `cheevd`

DOUBLE PRECISION for `zheevd`

Array, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix A in ascending order.
See also $info$.

a If $jobz = 'V'$, then on exit this array is overwritten by the unitary matrix Z which contains the eigenvectors of A .

$work(1)$ On exit, if $lwork > 0$, then the real part of $work(1)$ returns the required minimal size of $lwork$.

<i>rwork</i> (1)	On exit, if <i>lrwork</i> > 0, then <i>rwork</i> (1) returns the required minimal size of <i>lrwork</i> .
<i>iwork</i> (1)	On exit, if <i>liwork</i> > 0, then <i>iwork</i> (1) returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , and <i>jobz</i> = 'N', then the algorithm failed to converge; <i>i</i> off-diagonal elements of an intermediate tridiagonal form did not converge to zero; if <i>info</i> = <i>i</i> , and <i>jobz</i> = 'V', then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns <i>info</i> / (n+1) through mod(<i>info</i> , n+1). If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `heevd` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>w</i>	Holds the vector of length (<i>n</i>).
<i>jobz</i>	Must be 'N' or 'V'. The default value is 'N'.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix *A* + *E* such that $\|E\|_2 = O(\epsilon) * \|A\|_2$, where ϵ is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is `syevd`. See also `hpevd` for matrices held in packed storage, and `hbevd` for banded matrices.

?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax**FORTRAN 77:**

```
call ssyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work,
lwork, iwork, ifail, info)

call dsyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work,
lwork, iwork, ifail, info)
```

Fortran 95:

```
call syevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

C:

```
lapack_int LAPACKE_<?>syevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, <datatype>* a, lapack_int lda, <datatype> vl, <datatype> vu, lapack_int
il, lapack_int iu, <datatype> abstol, lapack_int* m, <datatype>* w, <datatype>* z,
lapack_int ldz, lapack_int* ifail );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of real symmetric eigenvalue problems the default choice should be [syevr](#) function as its underlying algorithm is faster and uses less workspace. ?syevx is faster for a few selected eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A', 'V', or 'I'. If <i>range</i> = 'A', all eigenvalues will be found. If <i>range</i> = 'V', all eigenvalues in the half-open interval $(vl, vu]$ will be found. If <i>range</i> = 'I', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.

If $uplo = 'U'$, a stores the upper triangular part of A .

If $uplo = 'L'$, a stores the lower triangular part of A .

n

INTEGER. The order of the matrix A ($n \geq 0$).

$a, work$

REAL for ssyevx

DOUBLE PRECISION for dsyevx.

Arrays:

$a(lda,*)$ is an array containing either upper or lower triangular part of the symmetric matrix A , as specified by $uplo$.

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The leading dimension of the array a . Must be at least $\max(1, n)$.

vl, vu

REAL for ssyevx

DOUBLE PRECISION for dsyevx.

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if $range = 'A'$ or $'I'$.

il, iu

INTEGER.

If $range = 'I'$, the indices of the smallest and largest eigenvalues to be returned.

Constraints: $1 \leq il \leq iu \leq n$, if $n > 0$;

$il = 1$ and $iu = 0$, if $n = 0$.

Not referenced if $range = 'A'$ or $'V'$.

$abstol$

REAL for ssyevx

DOUBLE PRECISION for dsyevx.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array z ; $ldz \geq 1$.

If $jobz = 'V'$, then $ldz \geq \max(1, n)$.

$lwork$

INTEGER.

The dimension of the array $work$.

If $n \leq 1$ then $lwork \geq 1$, otherwise $lwork=8*n$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

$iwork$

INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>m</i>	INTEGER. The total number of eigenvalues found; $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.
<i>w</i>	REAL for ssyevx DOUBLE PRECISION for dsyevx Array, DIMENSION at least $\max(1, n)$. The first <i>m</i> elements contain the selected eigenvalues of the matrix <i>A</i> in ascending order.
<i>z</i>	REAL for ssyevx DOUBLE PRECISION for dsyevx. Array <i>z</i> (<i>ldz</i> ,*) contains eigenvectors. The second dimension of <i>z</i> must be at least $\max(1, m)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> (<i>i</i>). If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array <i>z</i> ; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.
<i>work</i> (1)	On exit, if <i>lwork</i> > 0, then <i>work</i> (1) returns the required minimal size of <i>lwork</i> .
<i>ifail</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, then <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , then <i>i</i> eigenvectors failed to converge; their indices are stored in the array <i>ifail</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `syevx` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>w</code>	Holds the vector of length n .
<code>a</code>	Holds the matrix A of size (m, n) .
<code>ifail</code>	Holds the vector of length n .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>vl</code>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<code>vu</code>	Default value for this element is $vu = \text{HUGE}(vl)$.
<code>il</code>	Default value for this argument is $il = 1$.
<code>iu</code>	Default value for this argument is $iu = n$.
<code>abstol</code>	Default value for this element is $abstol = 0.0_{\text{WP}}$.
<code>jobz</code>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted Note that there will be an error condition if <code>ifail</code> is present and z is omitted.
<code>range</code>	Restored based on the presence of arguments vl , vu , il , iu as follows: $range = 'V'$, if one of or both vl and vu are present, $range = 'I'$, if one of or both il and iu are present, $range = 'A'$, if none of vl , vu , il , iu is present, Note that there will be an error condition if one of or both vl and vu are present and at the same time one of or both il and iu are present.

Application Notes

For optimum performance use $lwork \geq (nb+3) * n$, where nb is the maximum of the blocksize for `?sytrd` and `?ormtr` returned by `ilaenv`.

If it is not clear how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If $lwork$ has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If $lwork = -1$, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array `work`. This operation is called a workspace query.

Note that if $lwork$ is less than the minimal required value and is not equal to -1 , then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \varepsilon * \max(|a|, |b|)$, where ε is the machine precision.

If $abstol$ is less than or equal to zero, then $\epsilon^* |T|$ is used as tolerance, where $|T|$ is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues are computed most accurately when $abstol$ is set to twice the underflow threshold $2*\text{slamch('S')}$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2*\text{slamch('S')}$.

?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call cheevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work,
lwork, rwork, iwork, ifail, info)

call zheevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work,
lwork, rwork, iwork, ifail, info)
```

Fortran 95:

```
call heevx(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

C:

```
lapack_int LAPACKE_cheevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_float* a, lapack_int lda, float vl, float vu, lapack_int
il, lapack_int iu, float abstol, lapack_int* m, float* w, lapack_complex_float* z,
lapack_int ldz, lapack_int* ifail );

lapack_int LAPACKE_zheevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_double* a, lapack_int lda, double vl, double vu,
lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w,
lapack_complex_double* z, lapack_int ldz, lapack_int* ifail );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be `heevr` function as its underlying algorithm is faster and uses less workspace. `?heevx` is faster for a few selected eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'.
-------------	----------------------------------

If $jobz = 'N'$, then only eigenvalues are computed.

If $jobz = 'V'$, then eigenvalues and eigenvectors are computed.

range

CHARACTER*1. Must be 'A', 'V', or 'I'.

If $range = 'A'$, all eigenvalues will be found.

If $range = 'V'$, all eigenvalues in the half-open interval $(vl, vu]$ will be found.

If $range = 'I'$, the eigenvalues with indices il through iu will be found.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If $uplo = 'U'$, a stores the upper triangular part of A .

If $uplo = 'L'$, a stores the lower triangular part of A .

n

INTEGER. The order of the matrix A ($n \geq 0$).

a, work

COMPLEX for cheevx

DOUBLE COMPLEX for zheevx.

Arrays:

$a(ilda,*)$ is an array containing either upper or lower triangular part of the Hermitian matrix A , as specified by *uplo*.

The second dimension of a must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The leading dimension of the array a . Must be at least $\max(1, n)$.

vl, vu

REAL for cheevx

DOUBLE PRECISION for zheevx.

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if $range = 'A'$ or ' I '.

il, iu

INTEGER.

If $range = 'I'$, the indices of the smallest and largest eigenvalues to be returned. Constraints:

$1 \leq il \leq iu \leq n$, if $n > 0$; $il = 1$ and $iu = 0$, if $n = 0$. Not referenced if $range = 'A'$ or ' V '.

abstol

REAL for cheevx

DOUBLE PRECISION for zheevx. The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array z ; $ldz \geq 1$.

If $jobz = 'V'$, then $ldz \geq \max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

$lwork \geq 1$ if $n \leq 1$; otherwise at least $2 * n$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

See *Application Notes* for the suggested value of $lwork$.

$rwork$

REAL for `cheevx`

DOUBLE PRECISION for `zheevx`.

Workspace array, DIMENSION at least $\max(1, 7n)$.

$iwork$

INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

a

On exit, the lower triangle (if $uplo = 'L'$) or the upper triangle (if $uplo = 'U'$) of A , including the diagonal, is overwritten.

m

INTEGER. The total number of eigenvalues found; $0 \leq m \leq n$.

If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu-il+1$.

w

REAL for `cheevx`

DOUBLE PRECISION for `zheevx`

Array, DIMENSION at least $\max(1, n)$. The first m elements contain the selected eigenvalues of the matrix A in ascending order.

z

COMPLEX for `cheevx`

DOUBLE COMPLEX for `zheevx`.

Array $z(ldz,*)$ contains eigenvectors.

The second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

If $jobz = 'N'$, then z is not referenced. Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

$work(1)$

On exit, if $lwork > 0$, then $work(1)$ returns the required minimal size of $lwork$.

$ifail$

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, then $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'V'$, then $ifail$ is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array $ifail$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `heevx` interface are the following:

a	Holds the matrix A of size (n, n) .
w	Holds the vector of length n .
z	Holds the matrix Z of size (n, n) .
$ifail$	Holds the vector of length n .
$uplo$	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
vl	Default value for this element is $vl = -\text{HUGE}(v)$.
vu	Default value for this element is $vu = \text{HUGE}(v)$.
il	Default value for this argument is $il = 1$.
iu	Default value for this argument is $iu = n$.
$abstol$	Default value for this element is $abstol = 0.0_{\text{WP}}$.
$jobz$	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted Note that there will be an error condition if $ifail$ is present and z is omitted.
$range$	Restored based on the presence of arguments vl , vu , il , iu as follows: $range = 'V'$, if one of or both vl and vu are present, $range = 'I'$, if one of or both il and iu are present, $range = 'A'$, if none of vl , vu , il , iu is present, Note that there will be an error condition if one of or both vl and vu are present and at the same time one of or both il and iu are present.

Application Notes

For optimum performance use $\text{lwork} \geq (nb+1) * n$, where nb is the maximum of the blocksize for `?hetrd` and `?unmtr` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * \|T\|$ will be used in its place, where $\|T\|$ is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * \text{slamch('S')}$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * \text{slamch('S')}$.

?syevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using the Relatively Robust Representations.

Syntax

FORTRAN 77:

```
call ssyevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
isuppz, work, lwork, iwork, liwork, info)

call dsyevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
isuppz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call syevr(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

C:

```
lapack_int LAPACKE_<?>syevr( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, <datatype>* a, lapack_int lda, <datatype> vl, <datatype> vu, lapack_int
il, lapack_int iu, <datatype> abstol, lapack_int* m, <datatype>* w, <datatype>* z,
lapack_int ldz, lapack_int* isuppz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix *A*. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix A to tridiagonal form T with a call to [sytrd](#). Then, whenever possible, [?syevr](#) calls [stemr](#) to compute the eigenspectrum using Relatively Robust Representations. [stemr](#) computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L^*D^*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For each unreduced block of T :

- a. Compute $T - \sigma^* I = L^* D^* L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of D and L cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- b. Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).
- c. For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- d. For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine [?syevr](#) calls [stemr](#) when the full spectrum is requested on machines that conform to the IEEE-754 floating point standard. [?syevr](#) calls [stebz](#) and [stein](#) on non-IEEE machines and when partial spectrum requests are made.

Note that [?syevr](#) is preferable for most cases of real symmetric eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues <i>lambda(i)</i> in the half-open interval: $v1 < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> . For <i>range</i> = 'V' or 'I' and <i>iu-il</i> < <i>n-1</i> , <i>sstebz/dstebz</i> and <i>sstein/dstein</i> are called.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).

a, work

REAL for ssyevr
DOUBLE PRECISION for dsyevr.

Arrays:

a(*lda,**) is an array containing either upper or lower triangular part of the symmetric matrix *A*, as specified by *uplo*.

The second dimension of *a* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The leading dimension of the array *a*. Must be at least $\max(1, n)$.

vl, vu

REAL for ssyevr
DOUBLE PRECISION for dsyevr.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint:

$1 \leq il \leq iu \leq n$, if $n > 0$;
 $il=1$ and $iu=0$, if $n = 0$.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for ssyevr
DOUBLE PRECISION for dsyevr. The absolute error tolerance to which each eigenvalue/eigenvector is required.

If *jobz* = 'V', the eigenvalues and eigenvectors output have residual norms bounded by *abstol*, and the dot products between different eigenvectors are bounded by *abstol*.

If *abstol* < $n * \text{eps} * |T|$, then $n * \text{eps} * |T|$ is used instead, where *eps* is the machine precision, and $|T|$ is the 1-norm of the matrix *T*. The eigenvalues are computed to an accuracy of $\text{eps} * |T|$ irrespective of *abstol*.

If high relative accuracy is important, set *abstol* to ?lamch('S').

ldz

INTEGER. The leading dimension of the output array *z*.

Constraints:

$ldz \geq 1$ and
 $ldz \geq \max(1, n)$ if *jobz* = 'V'.

lwork

INTEGER.

The dimension of the array *work*.

Constraint: $lwork \geq \max(1, 26n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

$iwork$ INTEGER. Workspace array, its dimension $\max(1, liwork)$.

$liwork$ INTEGER.

The dimension of the array $iwork$, $lwork \geq \max(1, 10n)$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $iwork$ array, returns this value as the first entry of the $iwork$ array, and no error message related to $liwork$ is issued by [xerbla](#).

Output Parameters

a On exit, the lower triangle (if $uplo = 'L'$) or the upper triangle (if $uplo = 'U'$) of A , including the diagonal, is overwritten.

m INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.

If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu - il + 1$.

w, z REAL for [ssyevr](#)

DOUBLE PRECISION for [dsyevr](#).

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$, contains the selected eigenvalues in ascending order, stored in $w(1)$ to $w(m)$;

$z(ldz, *)$, the second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If $jobz = 'N'$, then z is not referenced. Note that you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

$isuppz$ INTEGER.

Array, DIMENSION at least $2 * \max(1, m)$.

The support of the eigenvectors in z , i.e., the indices indicating the nonzero elements in z . The i -th eigenvector is nonzero only in elements $isuppz(2i - 1)$ through $isuppz(2i)$. Referenced only if eigenvectors are needed ($jobz = 'V'$) and all eigenvalues are needed, that is, $range = 'A'$ or $range = 'I'$ and $il = 1$ and $iu = n$.

$work(1)$ On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

<i>iwork(1)</i>	On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the required minimal size of <i>lwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , an internal error has occurred.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `syevr` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>w</i>	Holds the vector of length <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>), where the values <i>n</i> and <i>m</i> are significant.
<i>isuppz</i>	Holds the vector of length (2*m), where the values (2*m) are significant.
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<i>vl</i>	Default value for this element is <i>vl</i> = - <code>HUGE(vl)</code> .
<i>vu</i>	Default value for this element is <i>vu</i> = <code>HUGE(vl)</code> .
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = ' <code>V</code> ', if <i>z</i> is present, <i>jobz</i> = ' <code>N</code> ', if <i>z</i> is omitted Note that there will be an error condition if <i>isuppz</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = ' <code>V</code> ', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = ' <code>I</code> ', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = ' <code>A</code> ', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

For optimum performance use *lwork* $\geq (nb+6)*n$, where *nb* is the maximum of the blocksize for `?sytrd` and `?ormtr` returned by `ilaenv`.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If $lwork$ (or $liwork$) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$) on exit. Use this value ($work(1)$, $iwork(1)$) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$). This operation is called a workspace query.

Note that if $lwork$ ($liwork$) is less than the minimal required value and is not equal to -1 , then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Normal execution of `?stegr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

?heevr

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using the Relatively Robust Representations.

Syntax

FORTRAN 77:

```
call cheevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
isuppz, work, lwork, rwork, lrwork, iwork, liwork, info)
call zheevr(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz,
isuppz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call heevr(a, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

C:

```
lapack_int LAPACKE_cheevr( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_float* a, lapack_int lda, float vl, float vu, lapack_int
il, lapack_int iu, float abstol, lapack_int* m, float* w, lapack_complex_float* z,
lapack_int ldz, lapack_int* isuppz );
lapack_int LAPACKE_zheevr( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_double* a, lapack_int lda, double vl, double vu,
lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w,
lapack_complex_double* z, lapack_int ldz, lapack_int* isuppz );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix A to tridiagonal form T with a call to `hetrd`. Then, whenever possible, `?heevr` calls `stegr` to compute the eigenspectrum using Relatively Robust Representations. `?stegr` computes eigenvalues by the `dqds` algorithm, while orthogonal eigenvectors are computed from various

"good" L^*D*L^T representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For each unreduced block (submatrix) of T :

- a. Compute $T - \sigma^* I = L^* D^* L^T$, so that L and D define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of D and L cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix T does not have this property in general.
- b. Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).
- c. For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- d. For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter `abstol`.

The routine `?heevr` calls `stemr` when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard, or `stebz` and `stein` on non-IEEE machines and when partial spectrum requests are made.

Note that the routine `?heevr` is preferable for most cases of complex Hermitian eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>jobz</code>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>V</code> '. If <code>job</code> = ' <code>N</code> ', then only eigenvalues are computed. If <code>job</code> = ' <code>V</code> ', then eigenvalues and eigenvectors are computed.
<code>range</code>	CHARACTER*1. Must be ' <code>A</code> ' or ' <code>V</code> ' or ' <code>I</code> '. If <code>range</code> = ' <code>A</code> ', the routine computes all eigenvalues. If <code>range</code> = ' <code>V</code> ', the routine computes eigenvalues <code>lambda(i)</code> in the half-open interval: $vl < \lambda(i) \leq vu$. If <code>range</code> = ' <code>I</code> ', the routine computes eigenvalues with indices <code>il</code> to <code>iu</code> . For <code>range</code> = ' <code>V</code> ' or ' <code>I</code> ', <code>sstebz/dstebz</code> and <code>cstein/zstein</code> are called.
<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '. If <code>uplo</code> = ' <code>U</code> ', <code>a</code> stores the upper triangular part of <code>A</code> . If <code>uplo</code> = ' <code>L</code> ', <code>a</code> stores the lower triangular part of <code>A</code> .
<code>n</code>	INTEGER. The order of the matrix <code>A</code> ($n \geq 0$).
<code>a, work</code>	COMPLEX for <code>cheevr</code> DOUBLE COMPLEX for <code>zheevr</code> . Arrays: <code>a(lda,*)</code> is an array containing either upper or lower triangular part of the Hermitian matrix <code>A</code> , as specified by <code>uplo</code> .

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The leading dimension of the array a .

Must be at least $\max(1, n)$.

vl, vu

REAL for cheevr

DOUBLE PRECISION for zheevr.

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If $range = 'A'$ or ' I' , vl and vu are not referenced.

il, iu

INTEGER.

If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.

If $range = 'A'$ or ' V' , il and iu are not referenced.

$abstol$

REAL for cheevr

DOUBLE PRECISION for zheevr.

The absolute error tolerance to which each eigenvalue/eigenvector is required.

If $jobz = 'V'$, the eigenvalues and eigenvectors output have residual norms bounded by $abstol$, and the dot products between different eigenvectors are bounded by $abstol$.

If $abstol < n * \text{eps} * |T|$, then $n * \text{eps} * |T|$ will be used in its place, where eps is the machine precision, and $|T|$ is the 1-norm of the matrix T . The eigenvalues are computed to an accuracy of $\text{eps} * |T|$ irrespective of $abstol$.

If high relative accuracy is important, set $abstol$ to `?lamch('S')`.

ldz

INTEGER. The leading dimension of the output array z . Constraints:

$ldz \geq 1$ if $jobz = 'N'$;

$ldz \geq \max(1, n)$ if $jobz = 'V'$.

$lwork$

INTEGER.

The dimension of the array $work$.

Constraint: $lwork \geq \max(1, 2n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lwork$ or $liwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

rwork REAL for cheevr
 DOUBLE PRECISION for zheevr.
 Workspace array, DIMENSION max(1, *lwork*).

lrwork INTEGER.
 The dimension of the array *rwork*;
 lwork $\geq \max(1, 24n)$.
 If *lrwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#).

iwork INTEGER. Workspace array, its dimension max(1, *liwork*).

liwork INTEGER.
 The dimension of the array *iwork*,
 lwork $\geq \max(1, 10n)$.
 If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#).

Output Parameters

a On exit, the lower triangle (if *uplo* = 'L') or the upper triangle (if *uplo* = 'U') of *A*, including the diagonal, is overwritten.

m INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$.
 If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w REAL for cheevr
 DOUBLE PRECISION for zheevr.
 Array, DIMENSION at least max(1, *n*), contains the selected eigenvalues in ascending order, stored in *w*(1) to *w*(*m*).

z COMPLEX for cheevr
 DOUBLE COMPLEX for zheevr.
 Array *z*(*ldz*, *); the second dimension of *z* must be at least max(1, *m*).
 If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).
 If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least $\max(1,m)$ columns are supplied in the array z ; if $\text{range} = \text{'V'}$, the exact value of m is not known in advance and an upper bound must be used.

isuppz

INTEGER.

Array, DIMENSION at least $2 * \max(1, m)$.

The support of the eigenvectors in z , i.e., the indices indicating the nonzero elements in z . The i -th eigenvector is nonzero only in elements $\text{isuppz}(2i-1)$ through $\text{isuppz}(2i)$. Referenced only if eigenvectors are needed ($\text{jobz} = \text{'V'}$) and all eigenvalues are needed, that is, $\text{range} = \text{'A'}$ or $\text{range} = \text{'I'}$ and $\text{il} = 1$ and $\text{iu} = n$.

work(1)

On exit, if $\text{info} = 0$, then $\text{work}(1)$ returns the required minimal size of lwork .

rwork(1)

On exit, if $\text{info} = 0$, then $\text{rwork}(1)$ returns the required minimal size of lrwork .

iwork(1)

On exit, if $\text{info} = 0$, then $\text{iwork}(1)$ returns the required minimal size of liwork .

info

INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th parameter had an illegal value.

If $\text{info} = i$, an internal error has occurred.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `heevr` interface are the following:

a

Holds the matrix A of size (n, n) .

w

Holds the vector of length n .

z

Holds the matrix Z of size (n, n) , where the values n and m are significant.

isuppz

Holds the vector of length $(2*n)$, where the values $(2*m)$ are significant.

uplo

Must be '`U`' or '`L`'. The default value is '`U`'.

vl

Default value for this element is $\text{vl} = -\text{HUGE}(\text{vl})$.

vu

Default value for this element is $\text{vu} = \text{HUGE}(\text{vl})$.

il

Default value for this argument is $\text{il} = 1$.

iu

Default value for this argument is $\text{iu} = n$.

abstol

Default value for this element is $\text{abstol} = 0.0_{\text{WP}}$.

<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if <i>isuppz</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

For optimum performance use *lwork* $\geq (nb+1)*n$, where *nb* is the maximum of the blocksize for ?hetrd and ?unmtr returned by ilaenv.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *lrwork*, or *liwork*) for the first run or set *lwork* = -1 (*lrwork* = -1, *liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *lrwork*, *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *rwork*, *iwork*) on exit. Use this value (*work*(1), *rwork*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *rwork*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*lrwork*, *liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Normal execution of ?steqr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

?spev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

FORTRAN 77:

```
call sspev(jobz, uplo, n, ap, w, z, ldz, work, info)
call dspev(jobz, uplo, n, ap, w, z, ldz, work, info)
```

Fortran 95:

```
call spev(ap, w [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_<?>spev( int matrix_layout, char jobz, char uplo, lapack_int n,
<datatype>* ap, <datatype>* w, <datatype>* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap, work</i>	REAL for sspev DOUBLE PRECISION for dspev Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of symmetric matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$.
<i>work</i>	(*) is a workspace array, DIMENSION at least $\max(1, 3n)$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then <i>ldz</i> ≥ 1 ; if <i>jobz</i> = 'V', then <i>ldz</i> $\geq \max(1, n)$.

Output Parameters

<i>w, z</i>	REAL for sspev DOUBLE PRECISION for dspev Arrays: <i>w</i> (*), DIMENSION at least $\max(1, n)$. If <i>info</i> = 0, <i>w</i> contains the eigenvalues of the matrix A in ascending order. <i>z</i> (<i>ldz</i> , *). The second dimension of <i>z</i> must be at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the orthonormal eigenvectors of the matrix A , with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> (<i>i</i>). If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
-------------	--

<i>ap</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spev` interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size (<i>n</i> *(<i>n</i> +1)/2).
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

?hpev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

FORTRAN 77:

```
call chpev(jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
call zhpev(jobz, uplo, n, ap, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hpev(ap, w [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_chpev( int matrix_layout, char jobz, char uplo, lapack_int n,
                           lapack_complex_float* ap, float* w, lapack_complex_float* z, lapack_int ldz );
lapack_int LAPACKE_zhpev( int matrix_layout, char jobz, char uplo, lapack_int n,
                           lapack_complex_double* ap, double* w, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90

- C: mkl.h

Description

The routine computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap, work</i>	COMPLEX for chpev DOUBLE COMPLEX for zhpev. Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of Hermitian matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. (*) is a workspace array, DIMENSION at least $\max(1, 2n-1)$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then <i>ldz</i> ≥ 1 ; if <i>jobz</i> = 'V', then <i>ldz</i> $\geq \max(1, n)$.
<i>rwork</i>	REAL for chpev DOUBLE PRECISION for zhpev. Workspace array, DIMENSION at least $\max(1, 3n-2)$.

Output Parameters

<i>w</i>	REAL for chpev DOUBLE PRECISION for zhpev. Array, DIMENSION at least $\max(1, n)$. If <i>info</i> = 0, <i>w</i> contains the eigenvalues of the matrix A in ascending order.
<i>z</i>	COMPLEX for chpev

DOUBLE COMPLEX for zhpev.

Array z ($ldz, *$).

The second dimension of z must be at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, z contains the orthonormal eigenvectors of the matrix A , with the i -th column of z holding the eigenvector associated with $w(i)$.

If $jobz = 'N'$, then z is not referenced.

ap

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A .

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpev` interface are the following:

ap Holds the array A of size $(n*(n+1)/2)$.

w Holds the vector with the number of elements n .

z Holds the matrix Z of size (n, n) .

$uplo$ Must be '`U`' or '`L`'. The default value is '`U`'.

$jobz$ Restored based on the presence of the argument z as follows:

$jobz = 'V'$, if z is present,

$jobz = 'N'$, if z is omitted.

?spevd

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix held in packed storage.

Syntax

FORTRAN 77:

```
call sspevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
```

```
call dspevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call spevd(ap, w [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_<?>spevd( int matrix_layout, char jobz, char uplo, lapack_int n,
<datatype>* ap, <datatype>* w, <datatype>* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A (held in packed storage). In other words, it can compute the spectral factorization of A as:

$$A = Z \Lambda Z^T.$$

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>V</code> '.
	If <i>jobz</i> = ' <code>N</code> ', then only eigenvalues are computed.
	If <i>jobz</i> = ' <code>V</code> ', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '.
	If <i>uplo</i> = ' <code>U</code> ', <i>ap</i> stores the packed upper triangular part of A .
	If <i>uplo</i> = ' <code>L</code> ', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap</i> , <i>work</i>	REAL for sspevd DOUBLE PRECISION for dspevd
	Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of symmetric matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be $\max(1, n*(n+1)/2)$ <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> .
	Constraints:

if $jobz = 'N'$, then $ldz \geq 1$;
 if $jobz = 'V'$, then $ldz \geq \max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

if $n \leq 1$, then $lwork \geq 1$;
 if $jobz = 'N'$ and $n > 1$, then $lwork \geq 2*n$;
 if $jobz = 'V'$ and $n > 1$, then
 $lwork \geq n^2 + 6*n + 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

*iwork*INTEGER. Workspace array, its dimension $\max(1, liwork)$.*liwork*

INTEGER.

The dimension of the array *iwork*.

Constraints:

if $n \leq 1$, then $liwork \geq 1$;
 if $jobz = 'N'$ and $n > 1$, then $liwork \geq 1$;
 if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w, z

REAL for sspevd

DOUBLE PRECISION for dspevd

Arrays:

w()*, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix *A* in ascending order.
 See also *info*.

z(ldz,)*.

The second dimension of *z* must be: at least 1 if $jobz = 'N'$; at least $\max(1, n)$ if $jobz = 'V'$.

If $jobz = 'V'$, then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *A*. If $jobz = 'N'$, then *z* is not referenced.

<i>ap</i>	On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.
<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required <i>lwork</i> .
<i>iwork(1)</i>	On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the required <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>i</i> , then the algorithm failed to converge; <i>i</i> indicates the number of elements of an intermediate tridiagonal form which did not converge to zero. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spevd` interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (n, n) .
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = ' <code>V</code> ', if <i>z</i> is present, <i>jobz</i> = ' <code>N</code> ', if <i>z</i> is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\varepsilon) * \|T\|_2$, where ε is the machine precision.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work(1)*, *iwork(1)*) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *lwork* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is `hpevd`.

See also `syevd` for matrices held in full storage, and `sbevd` for banded matrices.

?hpevd

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a complex Hermitian matrix held in packed storage.

Syntax**FORTRAN 77:**

```
call chpevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork,
info)

call zhpevd(jobz, uplo, n, ap, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork,
info)
```

Fortran 95:

```
call hpevd(ap, w [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_chpevd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_float* ap, float* w, lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhpevd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_double* ap, double* w, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix A (held in packed storage). In other words, it can compute the spectral factorization of A as: $A = Z^* \Lambda^* Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A .

If $uplo = 'L'$, ap stores the packed lower triangular part of A .

n INTEGER. The order of the matrix A ($n \geq 0$).

$ap, work$ COMPLEX for chpevd

DOUBLE COMPLEX for zhpevd

Arrays:

$ap(*)$ contains the packed upper or lower triangle of Hermitian matrix A , as specified by $uplo$.

The dimension of ap must be at least $\max(1, n*(n+1)/2)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

ldz INTEGER. The leading dimension of the output array z .

Constraints:

if $jobz = 'N'$, then $ldz \geq 1$;

if $jobz = 'V'$, then $ldz \geq \max(1, n)$.

$lwork$ INTEGER.

The dimension of the array $work$.

Constraints:

if $n \leq 1$, then $lwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $lwork \geq n$;

if $jobz = 'V'$ and $n > 1$, then $lwork \geq 2*n$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See [Application Notes](#) for details.

$rwork$ REAL for chpevd

DOUBLE PRECISION for zhpevd

Workspace array, its dimension $\max(1, lrwork)$.

$lrwork$ INTEGER.

The dimension of the array $rwork$. Constraints:

if $n \leq 1$, then $lrwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $lrwork \geq n$;

if $jobz = 'V'$ and $n > 1$, then $lrwork \geq 2*n^2 + 5*n + 1$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See [Application Notes](#) for details.

$iwork$ INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

if $n \leq 1$, then $liwork \geq 1$;if $jobz = 'N'$ and $n > 1$, then $liwork \geq 1$;if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

*w*REAL for `chpevd`DOUBLE PRECISION for `zhpevd`Array, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix *A* in ascending order.
See also *info*.

*z*COMPLEX for `chpevd`DOUBLE COMPLEX for `zhpevd`Array, DIMENSION (*ldz*,*).The second dimension of *z* must be:at least 1 if $jobz = 'N'$;at least $\max(1, n)$ if $jobz = 'V'$.

If $jobz = 'V'$, then this array is overwritten by the unitary matrix *Z* which contains the eigenvectors of *A*.

If $jobz = 'N'$, then *z* is not referenced.

ap

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

work(1)

On exit, if $info = 0$, then *work(1)* returns the required minimal size of *lwork*.

rwork(1)

On exit, if $info = 0$, then *rwork(1)* returns the required minimal size of *lrwork*.

iwork(1)

On exit, if $info = 0$, then *iwork(1)* returns the required minimal size of *liwork*.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpевд` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>w</code>	Holds the vector with the number of elements n .
<code>z</code>	Holds the matrix Z of size (n, n) .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>jobz</code>	Restored based on the presence of the argument <code>z</code> as follows: $jobz = 'V'$, if <code>z</code> is present, $jobz = 'N'$, if <code>z</code> is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of `lwork` (`liwork` or `lrwork`) for the first run or set `lwork = -1` (`liwork = -1`, `lrwork = -1`).

If you choose the first option and set any of admissible `lwork` (`liwork` or `lrwork`) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`) on exit. Use this value (`work(1)`, `iwork(1)`, `rwork(1)`) for subsequent runs.

If you set `lwork = -1` (`liwork = -1`, `lrwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`). This operation is called a workspace query.

Note that if you set `lwork` (`liwork`, `lrwork`) to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is [`spevd`](#).

See also [`heevd`](#) for matrices held in full storage, and [`hbевд`](#) for banded matrices.

?spevх

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

FORTRAN 77:

```
call sspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work,
iwork, ifail, info)

call dspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work,
iwork, ifail, info)
```

Fortran 95:

```
call spevx(ap, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

C:

```
lapack_int LAPACKE_<?>spevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, <datatype>* ap, <datatype> vl, <datatype> vu, lapack_int il, lapack_int
iu, <datatype> abstol, lapack_int* m, <datatype>* w, <datatype>* z, lapack_int ldz,
lapack_int* ifail );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap</i> , <i>work</i>	REAL for sspevx DOUBLE PRECISION for dspevx Arrays:

ap(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.

The dimension of *ap* must be at least $\max(1, n*(n+1)/2)$.

work(*) is a workspace array, DIMENSION at least $\max(1, 8n)$.

vl, vu

REAL for sspevx

DOUBLE PRECISION for dspevx

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: *vl* < *vu*.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu

INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; *il*=1 and *iu*=0

if $n = 0$.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol

REAL for sspevx

DOUBLE PRECISION for dspevx

The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldz

INTEGER. The leading dimension of the output array *z*.

Constraints:

if *jobz* = 'N', then *ldz* ≥ 1 ;

if *jobz* = 'V', then *ldz* $\geq \max(1, n)$.

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

ap

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of *A*.

m

INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$. If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu-il+1*.

w, z

REAL for sspevx

DOUBLE PRECISION for dspevx

Arrays:

w(*), DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the selected eigenvalues of the matrix A in ascending order.

$z(ldz,*)$.

The second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

If $jobz = 'N'$, then z is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

$ifail$

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array $ifail$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spevxx` interface are the following:

`ap` Holds the array A of size $(n*(n+1)/2)$.

`w` Holds the vector with the number of elements n .

`z` Holds the matrix Z of size (n, n) , where the values n and m are significant.

`ifail` Holds the vector with the number of elements n .

`uplo` Must be '`U`' or '`L`'. The default value is '`U`'.

`vl` Default value for this element is $vl = -\text{HUGE}(vl)$.

`vu` Default value for this element is $vu = \text{HUGE}(vl)$.

<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present. Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $\text{abstol} + \varepsilon * \max(|a|, |b|)$, where ε is the machine precision.

If *abstol* is less than or equal to zero, then $\varepsilon * ||T||_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * ?\text{lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * ?\text{lamch}('S')$.

?hpevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.

Syntax

FORTRAN 77:

```
call chpevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work,
rwork, iwork, ifail, info)

call zhpevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z, ldz, work,
rwork, iwork, ifail, info)
```

Fortran 95:

```
call hpevx(ap, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

C:

```
lapack_int LAPACKE_chpevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_float* ap, float vl, float vu, lapack_int il, lapack_int
iu, float abstol, lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz,
lapack_int* ifail );
```

```
lapack_int LAPACKE_zhpevx( int matrix_layout, char jobz, char range, char uplo,
                           lapack_int n, lapack_complex_double* ap, double vl, double vu, lapack_int il,
                           lapack_int iu, double abstol, lapack_int* m, double* w, lapack_complex_double* z,
                           lapack_int ldz, lapack_int* ifail );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ap</i> stores the packed upper triangular part of A . If <i>uplo</i> = 'L', <i>ap</i> stores the packed lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>ap, work</i>	COMPLEX for chpevx DOUBLE COMPLEX for zhpevx Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of the Hermitian matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>work</i> (*) is a workspace array, DIMENSION at least $\max(1, 2n)$.
<i>vl, vu</i>	REAL for chpevx DOUBLE PRECISION for zhpevx

If $\text{range} = \text{'V'}$, the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $\text{vl} < \text{vu}$.

If $\text{range} = \text{'A'}$ or 'I' , vl and vu are not referenced.

il, iu

INTEGER.

If $\text{range} = \text{'I'}$, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq \text{il} \leq \text{iu} \leq n$, if $n > 0$; $\text{il}=1$ and $\text{iu}=0$ if $n = 0$.

If $\text{range} = \text{'A'}$ or 'V' , il and iu are not referenced.

abstol

REAL for chpevx

DOUBLE PRECISION for zhpevx

The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldz

INTEGER. The leading dimension of the output array z .

Constraints:

if $\text{jobz} = \text{'N'}$, then $\text{ldz} \geq 1$;

if $\text{jobz} = \text{'V'}$, then $\text{ldz} \geq \max(1, n)$.

rwork

REAL for chpevx

DOUBLE PRECISION for zhpevx

Workspace array, DIMENSION at least $\max(1, 7n)$.

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

ap

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A .

m

INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.

If $\text{range} = \text{'A'}$, $m = n$, and if $\text{range} = \text{'I'}$, $m = \text{iu}-\text{il}+1$.

w

REAL for chpevx

DOUBLE PRECISION for zhpevx

Array, DIMENSION at least $\max(1, n)$.

If $\text{info} = 0$, contains the selected eigenvalues of the matrix A in ascending order.

z

COMPLEX for chpevx

DOUBLE COMPLEX for zhpevx

Array $\text{z}(\text{ldz}, *)$.

The second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

If $jobz = 'N'$, then z is not referenced.

Note: you must ensure that at least $\max(1,m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

$ifail$

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array $ifail$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpevxx` interface are the following:

ap	Holds the array A of size $(n*(n+1)/2)$.
w	Holds the vector with the number of elements n .
z	Holds the matrix Z of size (n, n) , where the values n and m are significant.
$ifail$	Holds the vector with the number of elements n .
$uplo$	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
vl	Default value for this element is $vl = -\text{HUGE}(vl)$.
vu	Default value for this element is $vu = \text{HUGE}(vl)$.
il	Default value for this argument is $il = 1$.
iu	Default value for this argument is $iu = n$.

<code>abstol</code>	Default value for this element is <code>abstol = 0.0_WP</code> .
<code>jobz</code>	Restored based on the presence of the argument <code>z</code> as follows: <code>jobz = 'V'</code> , if <code>z</code> is present, <code>jobz = 'N'</code> , if <code>z</code> is omitted Note that there will be an error condition if <code>ifail</code> is present and <code>z</code> is omitted.
<code>range</code>	Restored based on the presence of arguments <code>vl</code> , <code>vu</code> , <code>il</code> , <code>iu</code> as follows: <code>range = 'V'</code> , if one or both <code>vl</code> and <code>vu</code> are present, <code>range = 'I'</code> , if one or both <code>il</code> and <code>iu</code> are present, <code>range = 'A'</code> , if none of <code>vl</code> , <code>vu</code> , <code>il</code> , <code>iu</code> is present, Note that there will be an error condition if one of or both <code>vl</code> and <code>vu</code> are present and at the same time one of or both <code>il</code> and <code>iu</code> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $\text{abstol} + \varepsilon * \max(|a|, |b|)$, where ε is the machine precision.

If `abstol` is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when `abstol` is set to twice the underflow threshold $2 * ?lamch('S')$, not zero.

If this routine returns with `info > 0`, indicating that some eigenvectors did not converge, try setting `abstol` to $2 * ?lamch('S')$.

?sbev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

FORTRAN 77:

```
call ssbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
call dsbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, info)
```

Fortran 95:

```
call sbev(ab, w [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_<?>sbev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, <datatype>* ab, lapack_int ldab, <datatype>* w, <datatype>* z,
lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> (<i>n</i> \geq 0).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> (<i>kd</i> \geq 0).
<i>ab, work</i>	REAL for ssbev DOUBLE PRECISION for dsbev. Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 3n-2)$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least <i>kd</i> + 1.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then <i>ldz</i> \geq 1; if <i>jobz</i> = 'V', then <i>ldz</i> $\geq \max(1, n)$.

Output Parameters

<i>w, z</i>	REAL for ssbev DOUBLE PRECISION for dsbev Arrays: <i>w</i> (*), DIMENSION at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues of the matrix <i>A</i> in ascending order. <i>z</i> (<i>ldz</i> ,*). The second dimension of <i>z</i> must be at least $\max(1, n)$.
-------------	---

If $jobz = 'V'$, then if $info = 0$, z contains the orthonormal eigenvectors of the matrix A , with the i -th column of z holding the eigenvector associated with $w(i)$.

If $jobz = 'N'$, then z is not referenced.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

If $uplo = 'U'$, the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows kd and $kd+1$ of ab , and if $uplo = 'L'$, the diagonal and first subdiagonal of T are returned in the first two rows of ab .

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbev` interface are the following:

ab	Holds the array A of size $(kd+1, n)$.
w	Holds the vector with the number of elements n .
z	Holds the matrix Z of size (n, n) .
$uplo$	Must be ' U ' or ' L '. The default value is ' U '.
$jobz$	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted.

?hbev

Computes all eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

FORTRAN 77:

```
call chbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
call zhbev(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hbev(ab, w [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_chbev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, lapack_complex_float* ab, lapack_int ldab, float* w,
lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhbev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, lapack_complex_double* ab, lapack_int ldab, double* w,
lapack_complex_double* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab</i> , <i>work</i>	COMPLEX for chbev DOUBLE COMPLEX for zhbev. Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least $kd + 1$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then <i>ldz</i> ≥ 1 ;

if $\text{jobz} = \text{'V'}$, then $\text{ldz} \geq \max(1, n)$.

$rwork$

REAL for chbev

DOUBLE PRECISION for zhbev

Workspace array, DIMENSION at least $\max(1, 3n-2)$.

Output Parameters

w

REAL for chbev

DOUBLE PRECISION for zhbev

Array, DIMENSION at least $\max(1, n)$.

If $\text{info} = 0$, contains the eigenvalues in ascending order.

z

COMPLEX for chbev

DOUBLE COMPLEX for zhbev.

Array $z(\text{ldz}, *)$.

The second dimension of z must be at least $\max(1, n)$.

If $\text{jobz} = \text{'V'}$, then if $\text{info} = 0$, z contains the orthonormal eigenvectors of the matrix A , with the i -th column of z holding the eigenvector associated with $w(i)$.

If $\text{jobz} = \text{'N'}$, then z is not referenced.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

If $\text{uplo} = \text{'U'}$, the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows kd and $kd+1$ of ab , and if $\text{uplo} = \text{'L'}$, the diagonal and first subdiagonal of T are returned in the first two rows of ab .

$info$

INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i th parameter had an illegal value.

If $\text{info} = i$, then the algorithm failed to converge;

i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbev` interface are the following:

ab

Holds the array A of size $(kd+1, n)$.

w

Holds the vector with the number of elements n .

z

Holds the matrix Z of size (n, n) .

<i>uplo</i>	Must be ' <i>U</i> ' or ' <i>L</i> '. The default value is ' <i>U</i> '.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = ' <i>V</i> ', if <i>z</i> is present, <i>jobz</i> = ' <i>N</i> ', if <i>z</i> is omitted.

?sbevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric band matrix using divide and conquer algorithm.

Syntax**FORTRAN 77:**

```
call ssbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, iwork, liwork, info)
call dsbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call sbevd(ab, w [, uplo] [, z] [, info])
```

C:

```
lapack_int LAPACKE_<?>sbevd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, <datatype>* ab, lapack_int ldab, <datatype>* w, <datatype>* z,
lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric band matrix *A*. In other words, it can compute the spectral factorization of *A* as:

$$A = Z \Lambda Z^T$$

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$A^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the *QL* or *QR* algorithm.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
----------------------	---

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> ($kd \geq 0$).
<i>ab</i> , <i>work</i>	REAL for ssbevd DOUBLE PRECISION for dsbevd. Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, \text{lwork})$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least <i>kd</i> +1.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then <i>ldz</i> ≥ 1 ; if <i>jobz</i> = 'V', then <i>ldz</i> $\geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraints: if <i>n</i> ≤ 1 , then <i>lwork</i> ≥ 1 ; if <i>jobz</i> = 'N' and <i>n</i> > 1 , then <i>lwork</i> $\geq 2n$; if <i>jobz</i> = 'V' and <i>n</i> > 1 , then <i>lwork</i> $\geq 2*n^2 + 5*n + 1$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> and <i>iwork</i> arrays, returns these values as the first entries of the <i>work</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> or <i>liwork</i> is issued by xerbla . See <i>Application Notes</i> for details.
<i>iwork</i>	INTEGER. Workspace array, its dimension $\max(1, \text{liwork})$.
<i>liwork</i>	INTEGER. The dimension of the array <i>iwork</i> . Constraints: if <i>n</i> ≤ 1 , then <i>liwork</i> < 1 ; if <i>job</i> = 'N' and <i>n</i> > 1 , then <i>liwork</i> < 1 ; if <i>job</i> = 'V' and <i>n</i> > 1 , then <i>liwork</i> $< 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by [xerbla](#). See [Application Notes](#) for details.

Output Parameters

w, z

REAL for ssbevd

DOUBLE PRECISION for dsbevd

Arrays:

$w(*)$, size at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix A in ascending order.
See also $info$.

$z(ldz,*)$.

The second dimension of z must be:

at least 1 if $job = 'N'$;

at least $\max(1, n)$ if $job = 'V'$.

If $job = 'V'$, then this array is overwritten by the orthogonal matrix Z which contains the eigenvectors of A . The i -th column of Z contains the eigenvector which corresponds to the eigenvalue $w(i)$.

If $job = 'N'$, then z is not referenced.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

$work(1)$

On exit, if $lwork > 0$, then $work(1)$ returns the required minimal size of $lwork$.

$iwork(1)$

On exit, if $liwork > 0$, then $iwork(1)$ returns the required minimal size of $liwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbevd` interface are the following:

ab

Holds the array A of size $(kd+1,n)$.

w

Holds the vector with the number of elements n .

<i>z</i>	Holds the matrix Z of size (n, n) .
<i>uplo</i>	Must be ' <i>U</i> ' or ' <i>L</i> '. The default value is ' <i>U</i> '.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = ' <i>V</i> ', if <i>z</i> is present, <i>jobz</i> = ' <i>N</i> ', if <i>z</i> is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If any of admissible *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *work* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The complex analogue of this routine is [hbevd](#).

See also [syevd](#) for matrices held in full storage, and [spevd](#) for matrices held in packed storage.

?hbевd

Computes all eigenvalues and (optionally) all eigenvectors of a complex Hermitian band matrix using divide and conquer algorithm.

Syntax

FORTRAN 77:

```
call chbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
call zhbevd(jobz, uplo, n, kd, ab, ldab, w, z, ldz, work, lwork, rwork, lrwork, iwork, liwork, info)
```

Fortran 95:

```
call hbевd(ab, w [, uplo] [, z] [, info])
```

C:

```
lapack_int LAPACKE_chbevd( int matrix_layout, char jobz, char uplo, lapack_int n,
                           lapack_int kd, lapack_complex_float* ab, lapack_int ldab, float* w,
                           lapack_complex_float* z, lapack_int ldz );
lapack_int LAPACKE_zhbevd( int matrix_layout, char jobz, char uplo, lapack_int n,
                           lapack_int kd, lapack_complex_double* ab, lapack_int ldab, double* w,
                           lapack_complex_double* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian band matrix A . In other words, it can compute the spectral factorization of A as: $A = Z^* \Lambda^* Z^H$.

Here Λ is a real diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the (complex) unitary matrix whose columns are the eigenvectors z_i . Thus,

$$A^* z_i = \lambda_i^* z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> (<i>n</i> \geq 0).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> (<i>kd</i> \geq 0).
<i>ab</i> , <i>work</i>	COMPLEX for chbevd DOUBLE COMPLEX for zhbevd. Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, its dimension $\max(1, lwork)$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least <i>kd</i> +1.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> . Constraints: if <i>jobz</i> = 'N', then <i>ldz</i> \geq 1;

if $jobz = 'V'$, then $ldz \geq \max(1, n)$.

$lwork$

INTEGER.

The dimension of the array $work$.

Constraints:

if $n \leq 1$, then $lwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $lwork \geq n$;

if $jobz = 'V'$ and $n > 1$, then $lwork \geq 2*n^2$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

$rwork$

REAL for chbevd

DOUBLE PRECISION for zhbevd

Workspace array, DIMENSION at least $lrwork$.

$lrwork$

INTEGER.

The dimension of the array $rwork$.

Constraints:

if $n \leq 1$, then $lrwork \geq 1$;

if $jobz = 'N'$ and $n > 1$, then $lrwork \geq n$;

if $jobz = 'V'$ and $n > 1$, then $lrwork \geq 2*n^2 + 5*n + 1$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

$iwork$

INTEGER. Workspace array, DIMENSION $\max(1, liwork)$.

$liwork$

INTEGER.

The dimension of the array $iwork$.

Constraints:

if $jobz = 'N'$ or $n \leq 1$, then $liwork \geq 1$;

if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

w

REAL for chbevd

DOUBLE PRECISION for zhbevd

Array, DIMENSION at least max(1, n).

If $info = 0$, contains the eigenvalues of the matrix A in ascending order.
See also $info$.

z

COMPLEX for chbevd

DOUBLE COMPLEX for zhbevd

Array, DIMENSION ($ldz, *$).

The second dimension of z must be:

at least 1 if $jobz = 'N'$;

at least max(1, n) if $jobz = 'V'$.

If $jobz = 'V'$, then this array is overwritten by the unitary matrix Z which contains the eigenvectors of A . The i -th column of Z contains the eigenvector which corresponds to the eigenvalue $w(i)$.

If $jobz = 'N'$, then z is not referenced.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

$work(1)$

On exit, if $lwork > 0$, then the real part of $work(1)$ returns the required minimal size of $lwork$.

$rwork(1)$

On exit, if $lrwork > 0$, then $rwork(1)$ returns the required minimal size of $lrwork$.

$iwork(1)$

On exit, if $liwork > 0$, then $iwork(1)$ returns the required minimal size of $liwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If $info = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbevd` interface are the following:

ab Holds the array A of size $(kd+1, n)$.

w Holds the vector with the number of elements n .

z Holds the matrix Z of size (n, n) .

$uplo$ Must be '`U`' or '`L`'. The default value is '`U`'.

jobz Restored based on the presence of the argument *z* as follows:
jobz = 'V', if *z* is present,
jobz = 'N', if *z* is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T + E$ such that $\|E\|_2 = O(\epsilon) \|T\|_2$, where ϵ is the machine precision.

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work*(1), *iwork*(1), *rwork*(1)) for subsequent runs.

If you set *lwork* = -1 (*liwork* = -1, *lrwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*, *lrwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The real analogue of this routine is [sbevd](#).

See also [heevd](#) for matrices held in full storage, and [hpevd](#) for matrices held in packed storage.

?sbevx

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

FORTRAN 77:

```
call ssbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w,
z, ldz, work, iwork, ifail, info)
call dsbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w,
z, ldz, work, iwork, ifail, info)
```

Fortran 95:

```
call sbevx(ab, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q] [,abstol]
[,info])
```

C:

```
lapack_int LAPACKE_<?>sbevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int kd, <datatype>* ab, lapack_int ldab, <datatype>* q,
lapack_int ldq, <datatype> vl, <datatype> vu, lapack_int il, lapack_int iu, <datatype>
abstol, lapack_int* m, <datatype>* w, <datatype>* z, lapack_int ldz, lapack_int*
ifail );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90

- C: mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be ' <i>N</i> ' or ' <i>V</i> '. If <i>jobz</i> = ' <i>N</i> ', then only eigenvalues are computed. If <i>jobz</i> = ' <i>V</i> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <i>A</i> ' or ' <i>V</i> ' or ' <i>I</i> '. If <i>range</i> = ' <i>A</i> ', the routine computes all eigenvalues. If <i>range</i> = ' <i>V</i> ', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = ' <i>I</i> ', the routine computes eigenvalues in range <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '. If <i>uplo</i> = ' <i>U</i> ', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = ' <i>L</i> ', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> (<i>n</i> \geq 0).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> (<i>kd</i> \geq 0).
<i>ab, work</i>	REAL for ssbevx DOUBLE PRECISION for dsbevx. Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(1, 7n)$.
<i>ldab</i>	INTEGER. The leading dimension of <i>ab</i> ; must be at least <i>kd</i> + 1.
<i>vl, vu</i>	REAL for ssbevx DOUBLE PRECISION for dsbevx. If <i>range</i> = ' <i>V</i> ', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: <i>vl</i> < <i>vu</i> .

If $\text{range} = \text{'A'}$ or 'I' , vl and vu are not referenced.

il, iu

INTEGER.

If $\text{range} = \text{'I'}$, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq \text{il} \leq \text{iu} \leq n$, if $n > 0$; $\text{il}=1$ and $\text{iu}=0$ if $n = 0$.

If $\text{range} = \text{'A'}$ or 'V' , il and iu are not referenced.

abstol

REAL for chpevx

DOUBLE PRECISION for zhpevx

The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldq, ldz

INTEGER. The leading dimensions of the output arrays q and z , respectively.

Constraints:

$\text{ldq} \geq 1$, $\text{ldz} \geq 1$;

If $\text{jobz} = \text{'V'}$, then $\text{ldq} \geq \max(1, n)$ and $\text{ldz} \geq \max(1, n)$.

iwork

INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

q

REAL for ssbevx DOUBLE PRECISION for dsbevx .

Array, DIMENSION (ldz, n).

If $\text{jobz} = \text{'V'}$, the n -by- n orthogonal matrix is used in the reduction to tridiagonal form.

If $\text{jobz} = \text{'N'}$, the array q is not referenced.

m

INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.

If $\text{range} = \text{'A'}$, $m = n$, and if $\text{range} = \text{'I'}$, $m = \text{iu}-\text{il}+1$.

w, z

REAL for ssbevx

DOUBLE PRECISION for dsbevx

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$. The first m elements of w contain the selected eigenvalues of the matrix A in ascending order.

$z(\text{ldz}, *)$.

The second dimension of z must be at least $\max(1, m)$.

If $\text{jobz} = \text{'V'}$, then if $\text{info} = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

If $jobz = 'N'$, then z is not referenced.

Note: you must ensure that at least $\max(1,m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

If $uplo = 'U'$, the first superdiagonal and the diagonal of the tridiagonal matrix T are returned in rows kd and $kd+1$ of ab , and if $uplo = 'L'$, the diagonal and first subdiagonal of T are returned in the first two rows of ab .

$ifail$

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array $ifail$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbevxx` interface are the following:

ab Holds the array A of size $(kd+1, n)$.

w Holds the vector with the number of elements n .

z Holds the matrix Z of size (n, n) , where the values n and m are significant.

$ifail$ Holds the vector with the number of elements n .

q Holds the matrix Q of size (n, n) .

$uplo$ Must be ' U ' or ' L '. The default value is ' U '.

$v1$ Default value for this element is $v1 = -\text{HUGE}(v1)$.

vu Default value for this element is $vu = \text{HUGE}(v1)$.

il Default value for this argument is $il = 1$.

<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if either <i>ifail</i> or <i>q</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \varepsilon \max(|a|, |b|)$, where ε is the machine precision.

If *abstol* is less than or equal to zero, then $\varepsilon^* \|T\|_1$ is used as tolerance, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2^*?lamch('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2^*?lamch('S')$.

?hbevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

Syntax

FORTRAN 77:

```
call chbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w,
z, ldz, work, rwork, iwork, ifail, info)
call zhbevx(jobz, range, uplo, n, kd, ab, ldab, q, ldq, vl, vu, il, iu, abstol, m, w,
z, ldz, work, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hbevx(ab, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q] [,abstol]
[,info])
```

C:

```
lapack_int LAPACKE_chbevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int kd, lapack_complex_float* ab, lapack_int ldab,
lapack_complex_float* q, lapack_int ldq, float vl, float vu, lapack_int il, lapack_int
iu, float abstol, lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz,
lapack_int* ifail );
```

```
lapack_int LAPACKE_zhbevx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int kd, lapack_complex_double* ab, lapack_int ldab,
lapack_complex_double* q, lapack_int ldq, double vl, double vu, lapack_int il,
lapack_int iu, double abstol, lapack_int* m, double* w, lapack_complex_double* z,
lapack_int ldz, lapack_int* ifail );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be ' <i>N</i> ' or ' <i>V</i> '. If <i>job</i> = ' <i>N</i> ', then only eigenvalues are computed. If <i>job</i> = ' <i>V</i> ', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be ' <i>A</i> ' or ' <i>V</i> ' or ' <i>I</i> '. If <i>range</i> = ' <i>A</i> ', the routine computes all eigenvalues. If <i>range</i> = ' <i>V</i> ', the routine computes eigenvalues <i>lambda(i)</i> in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = ' <i>I</i> ', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be ' <i>U</i> ' or ' <i>L</i> '. If <i>uplo</i> = ' <i>U</i> ', <i>ab</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = ' <i>L</i> ', <i>ab</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	INTEGER. The order of the matrix <i>A</i> (<i>n</i> ≥ 0).
<i>kd</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> (<i>kd</i> ≥ 0).
<i>ab</i> , <i>work</i>	COMPLEX for chbevx DOUBLE COMPLEX for zhbevx. Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of <i>ab</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array.

The dimension of *work* must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of *ab*; must be at least $kd + 1$.

vl, *REAL for chbevx*

DOUBLE PRECISION for zhbevx.

If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, *iu* INTEGER.

If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.

If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol REAL for chbevx

DOUBLE PRECISION for zhbevx.

The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldq, *ldz* INTEGER. The leading dimensions of the output arrays *q* and *z*, respectively.

Constraints:

$ldq \geq 1$, $ldz \geq 1$;

If *jobz* = 'V', then $ldq \geq \max(1, n)$ and $ldz \geq \max(1, n)$.

rwork REAL for chbevx

DOUBLE PRECISION for zhbevx

Workspace array, DIMENSION at least $\max(1, 7n)$.

iwork INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

q COMPLEX for chbevx DOUBLE COMPLEX for zhbevx.

Array, DIMENSION (*ldz*,*n*).

If *jobz* = 'V', the *n*-by-*n* unitary matrix is used in the reduction to tridiagonal form.

If *jobz* = 'N', the array *q* is not referenced.

m INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$.

If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu*-*il*+1.

w REAL for chbevx

DOUBLE PRECISION for zhbevx

Array, DIMENSION at least max(1, n). The first m elements contain the selected eigenvalues of the matrix A in ascending order.

z

COMPLEX for chbevx

DOUBLE COMPLEX for zhbevx.

Array *z*(*ldz*,*).

The second dimension of *z* must be at least max(1, *m*).

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*).

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

Note: you must ensure that at least max(1,*m*) columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and an upper bound must be used.

ab

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

If *uplo* = 'U', the first superdiagonal and the diagonal of the tridiagonal matrix *T* are returned in rows *kd* and *kd*+1 of *ab*, and if *uplo* = 'L', the diagonal and first subdiagonal of *T* are returned in the first two rows of *ab*.

ifail

INTEGER.

Array, DIMENSION at least max(1, n).

If *jobz* = 'V', then if *info* = 0, the first *m* elements of *ifail* are zero; if *info* > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If *jobz* = 'N', then *ifail* is not referenced.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then *i* eigenvectors failed to converge; their indices are stored in the array *ifail*.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbevx` interface are the following:

ab

Holds the array *A* of size (*kd*+1, *n*).

<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>), where the values <i>n</i> and <i>m</i> are significant.
<i>ifail</i>	Holds the vector with the number of elements <i>n</i> .
<i>q</i>	Holds the matrix <i>Q</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is <i>vl</i> = -HUGE(<i>vl</i>).
<i>vu</i>	Default value for this element is <i>vu</i> = HUGE(<i>vl</i>).
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if either <i>ifail</i> or <i>q</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present. Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to *abstol* + $\varepsilon * \max(|a|, |b|)$, where ε is the machine precision.

If *abstol* is less than or equal to zero, then $\varepsilon * ||T||_1$ will be used in its place, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * ?lamch('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * ?lamch('S')$.

?stev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.

Syntax

FORTRAN 77:

```
call sstev(jobz, n, d, e, z, ldz, work, info)
call dstev(jobz, n, d, e, z, ldz, work, info)
```

Fortran 95:

```
call stev(d, e [,z] [,info])
```

C:

```
lapack_int LAPACKE_<?>stev( int matrix_layout, char jobz, lapack_int n, <datatype>* d,
<datatype>* e, <datatype>* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>d, e, work</i>	REAL for ssstev DOUBLE PRECISION for dstev. Arrays: $d(*)$ contains the n diagonal elements of the tridiagonal matrix A . The dimension of d must be at least $\max(1, n)$. $e(*)$ contains the $n-1$ subdiagonal elements of the tridiagonal matrix A . The dimension of e must be at least $\max(1, n-1)$. The n -th element of this array is used as workspace. $work(*)$ is a workspace array. The dimension of $work$ must be at least $\max(1, 2n-2)$. If <i>jobz</i> = 'N', $work$ is not referenced.
<i>ldz</i>	INTEGER. The leading dimension of the output array z ; $ldz \geq 1$. If <i>jobz</i> = 'V' then $ldz \geq \max(1, n)$.

Output Parameters

<i>d</i>	On exit, if <i>info</i> = 0, contains the eigenvalues of the matrix A in ascending order.
<i>z</i>	REAL for ssstev DOUBLE PRECISION for dstev

`Array, DIMENSION (ldz, *).`

The second dimension of *z* must be at least $\max(1, n)$.

If *jobz* = 'V', then if *info* = 0, *z* contains the orthonormal eigenvectors of the matrix *A*, with the *i*-th column of *z* holding the eigenvector associated with the eigenvalue returned in *d*(*i*).

If *job* = 'N', then *z* is not referenced.

e

On exit, this array is overwritten with intermediate results.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, then the algorithm failed to converge;

i elements of *e* did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stev` interface are the following:

d Holds the vector of length *n*.

e Holds the vector of length *n*.

z Holds the matrix *Z* of size (*n*, *n*).

jobz Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,

jobz = 'N', if *z* is omitted.

?stevd

Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.

Syntax

FORTRAN 77:

```
call sstevd(jobz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
```

```
call dstevd(jobz, n, d, e, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call stevd(d, e [, z] [, info])
```

C:

```
lapack_int LAPACKE_<?>stevd( int matrix_layout, char jobz, lapack_int n, <datatype>* d, <datatype>* e, <datatype>* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric tridiagonal matrix T . In other words, the routine can compute the spectral factorization of T as: $T = Z^* \Lambda * Z^T$.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i , and Z is the orthogonal matrix whose columns are the eigenvectors z_i . Thus,

$$T^* z_i = \lambda_i * z_i \text{ for } i = 1, 2, \dots, n.$$

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

There is no complex analogue of this routine.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>d, e, work</i>	REAL for sstevd DOUBLE PRECISION for dstevd. Arrays: $d(*)$ contains the n diagonal elements of the tridiagonal matrix T . The dimension of d must be at least $\max(1, n)$. $e(*)$ contains the $n-1$ off-diagonal elements of T . The dimension of e must be at least $\max(1, n-1)$. The n -th element of this array is used as workspace. $work(*)$ is a workspace array. The dimension of $work$ must be at least $lwork$.
<i>ldz</i>	INTEGER. The leading dimension of the output array z . Constraints: $ldz \geq 1$ if <i>job</i> = 'N'; $ldz < \max(1, n)$ if <i>job</i> = 'V'.
<i>lwork</i>	INTEGER. The dimension of the array $work$. Constraints:

if $jobz = 'N'$ or $n \leq 1$, then $lwork \geq 1$;
 if $jobz = 'V'$ and $n > 1$, then $lwork \geq n^2 + 4*n + 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER. Workspace array, its dimension $\max(1, liwork)$.

liwork INTEGER.

The dimension of the array *iwork*.

Constraints:

if $jobz = 'N'$ or $n \leq 1$, then $liwork \geq 1$;
 if $jobz = 'V'$ and $n > 1$, then $liwork \geq 5*n+3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

d On exit, if $info = 0$, contains the eigenvalues of the matrix *T* in ascending order.

See also *info*.

z REAL for *sstevd*

DOUBLE PRECISION for *dstevd*

Array, DIMENSION (*ldz*, *).

The second dimension of *z* must be:

at least 1 if $jobz = 'N'$;

at least $\max(1, n)$ if $jobz = 'V'$.

If $jobz = 'V'$, then this array is overwritten by the orthogonal matrix *Z* which contains the eigenvectors of *T*.

If $jobz = 'N'$, then *z* is not referenced.

e On exit, this array is overwritten with intermediate results.

work(1) On exit, if $lwork > 0$, then *work*(1) returns the required minimal size of *lwork*.

iwork(1) On exit, if $liwork > 0$, then *iwork*(1) returns the required minimal size of *liwork*.

info INTEGER.

If $info = 0$, the execution is successful.

If $\text{info} = i$, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

If $\text{info} = -i$, the i -th parameter had an illegal value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stevd` interface are the following:

d	Holds the vector of length n .
e	Holds the vector of length n .
z	Holds the matrix Z of size (n, n) .
jobz	Restored based on the presence of the argument z as follows: $\text{jobz} = 'V'$, if z is present, $\text{jobz} = 'N'$, if z is omitted.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $\|E\|_2 = O(\epsilon) * \|T\|_2$, where ϵ is the machine precision.

If λ_i is an exact eigenvalue, and m_i is the corresponding computed value, then

$$|\mu_i - \lambda_i| \leq c(n) * \epsilon * \|T\|_2$$

where $c(n)$ is a modestly increasing function of n .

If z_i is the corresponding exact eigenvector, and w_i is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n) * \epsilon * \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.$$

Thus the accuracy of a computed eigenvector depends on the gap between its eigenvalue and all the other eigenvalues.

If it is not clear how much workspace to supply, use a generous value of lwork (or liwork) for the first run, or set $\text{lwork} = -1$ ($\text{liwork} = -1$).

If lwork (or liwork) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (work , iwork) on exit. Use this value ($\text{work}(1)$, $\text{iwork}(1)$) for subsequent runs.

If $\text{lwork} = -1$ ($\text{liwork} = -1$), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (work , iwork). This operation is called a workspace query.

Note that if lwork (liwork) is less than the minimal required value and is not equal to -1 , then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?stevx

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax**FORTRAN 77:**

```
call sstevx(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork,
ifail, info)

call dstevx(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, work, iwork,
ifail, info)
```

Fortran 95:

```
call stevx(d, e, w [, z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol] [,info])
```

C:

```
lapack_int LAPACKE_<?>stevx( int matrix_layout, char jobz, char range, lapack_int n,
<datatype>* d, <datatype>* e, <datatype> vl, <datatype> vu, lapack_int il, lapack_int
iu, <datatype> abstol, lapack_int* m, <datatype>* w, <datatype>* z, lapack_int ldz,
lapack_int* ifail );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix A . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>job</i> = 'N', then only eigenvalues are computed. If <i>job</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>d, e, work</i>	REAL for sstevx DOUBLE PRECISION for dstevx.

Arrays:

$d(*)$ contains the n diagonal elements of the tridiagonal matrix A .

The dimension of d must be at least $\max(1, n)$.

$e(*)$ contains the $n-1$ subdiagonal elements of A .

The dimension of e must be at least $\max(1, n-1)$. The n -th element of this array is used as workspace.

$work(*)$ is a workspace array.

The dimension of $work$ must be at least $\max(1, 5n)$.

vl, vu

REAL for sstevx

DOUBLE PRECISION for dstevx.

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If $range = 'A'$ or ' I' , vl and vu are not referenced.

il, iu

INTEGER.

If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.

If $range = 'A'$ or ' V' , il and iu are not referenced.

$abstol$

REAL for sstevx

DOUBLE PRECISION for dstevx. The absolute error tolerance to which each eigenvalue is required. See *Application notes* for details on error tolerance.

ldz

INTEGER. The leading dimensions of the output array z ; $ldz \geq 1$. If $jobz = 'V'$, then $ldz \geq \max(1, n)$.

$iwork$

INTEGER. Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

m

INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$.

If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu-il+1$.

w, z

REAL for sstevx

DOUBLE PRECISION for dstevx.

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$.

The first m elements of w contain the selected eigenvalues of the matrix A in ascending order.

$z(ldz, *)$.

The second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.

If $jobz = 'N'$, then z is not referenced.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

d, e

On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

$ifail$

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then i eigenvectors failed to converge; their indices are stored in the array $ifail$.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stevx` interface are the following:

d	Holds the vector of length n .
e	Holds the vector of length n .
w	Holds the vector of length n .
z	Holds the matrix Z of size (n, n) , where the values n and m are significant.
$ifail$	Holds the vector of length n .
vl	Default value for this element is $vl = -\text{HUGE}(vl)$.
vu	Default value for this element is $vu = \text{HUGE}(vl)$.
il	Default value for this argument is $il = 1$.

<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $\text{abstol} + \varepsilon * \max(|a|, |b|)$, where ε is the machine precision.

If *abstol* is less than or equal to zero, then $\varepsilon * ||A||^1$ is used instead. Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold $2 * ?\text{lamch('S')}$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, set *abstol* to $2 * ?\text{lamch('S')}$.

?stevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

Syntax

FORTRAN 77:

```
call sstevr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)

call dstevr(jobz, range, n, d, e, vl, vu, il, iu, abstol, m, w, z, ldz, isuppz, work,
lwork, iwork, liwork, info)
```

Fortran 95:

```
call stevr(d, e, w [, z] [,vl] [,vu] [,il] [,iu] [,m] [,isuppz] [,abstol] [,info])
```

C:

```
lapack_int LAPACKE_<?>stevr( int matrix_layout, char jobz, char range, lapack_int n,
<datatype>* d, <datatype>* e, <datatype> vl, <datatype> vu, lapack_int il, lapack_int
iu, <datatype> abstol, lapack_int* m, <datatype>* w, <datatype>* z, lapack_int ldz,
lapack_int* isuppz );
```

Include Files

- Fortran: mkl.fi

- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T . Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, the routine calls [stemr](#) to compute the eigenspectrum using Relatively Robust Representations. [stegr](#) computes eigenvalues by the *dqds* algorithm, while orthogonal eigenvectors are computed from various "good" $L^*D^*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the i -th unreduced block of T :

- Compute $T - \sigma_i = L_i * D_i * L_i^T$, such that $L_i * D_i * L_i^T$ is a relatively robust representation.
- Compute the eigenvalues, λ_j , of $L_i * D_i * L_i^T$ to high relative accuracy by the *dqds* algorithm.
- If there is a cluster of close eigenvalues, "choose" σ_i close to the cluster, and go to Step (a).
- Given the approximate eigenvalue λ_j of $L_i * D_i * L_i^T$, compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter *abstol*.

The routine [?stevr](#) calls [stemr](#) when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. [?stevr](#) calls [stebz](#) and [stein](#) on non-IEEE machines and when partial spectrum requests are made.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues <i>lambda</i> (<i>i</i>) in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> . For <i>range</i> = 'V' or 'I' and <i>iu-il</i> < <i>n-1</i> , sstebz/dstebz and sstein/dstein are called.
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>d, e, work</i>	REAL for sstevr DOUBLE PRECISION for dstevr . Arrays: <i>d</i> (*) contains the n diagonal elements of the tridiagonal matrix T . The dimension of <i>d</i> must be at least $\max(1, n)$.

$e(*)$ contains the $n-1$ subdiagonal elements of A .

The dimension of e must be at least $\max(1, n-1)$. The n -th element of this array is used as workspace.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

vl, vu

REAL for sstevr

DOUBLE PRECISION for dstevr.

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If $range = 'A'$ or ' I' , vl and vu are not referenced.

il, iu

INTEGER.

If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.

If $range = 'A'$ or ' V' , il and iu are not referenced.

$abstol$

REAL for sstevr

DOUBLE PRECISION for dstevr.

The absolute error tolerance to which each eigenvalue/eigenvector is required.

If $jobz = 'V'$, the eigenvalues and eigenvectors output have residual norms bounded by $abstol$, and the dot products between different eigenvectors are bounded by $abstol$. If $abstol < n * \text{eps} * |T|$, then $n * \text{eps} * |T|$ will be used in its place, where eps is the machine precision, and $|T|$ is the 1-norm of the matrix T . The eigenvalues are computed to an accuracy of $\text{eps} * |T|$ irrespective of $abstol$.

If high relative accuracy is important, set $abstol$ to $?lamch('S')$.

ldz

INTEGER. The leading dimension of the output array z .

Constraints:

$ldz \geq 1$ if $jobz = 'N'$;

$ldz \geq \max(1, n)$ if $jobz = 'V'$.

$lwork$

INTEGER.

The dimension of the array $work$. Constraint:

$lwork \geq \max(1, 20*n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $iwork$ is issued by [xerbla](#). See *Application Notes* for details.

$iwork$

INTEGER.

Workspace array, its dimension $\max(1, liwork)$.

$liwork$

INTEGER.

The dimension of the array $iwork$,

$iwork \geq \max(1, 10^*n)$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

m

INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$. If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = iu-il+1$.

w, z

REAL for `sstevr`

DOUBLE PRECISION for `dstevr`.

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$.

The first m elements of w contain the selected eigenvalues of the matrix T in ascending order.

$z(ldz,*)$.

The second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

If $jobz = 'N'$, then z is not referenced.

Note: you must ensure that at least $\max(1,m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

d, e

On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

$isuppz$

INTEGER.

Array, DIMENSION at least $2 * \max(1, m)$.

The support of the eigenvectors in z , i.e., the indices indicating the nonzero elements in z . The i -th eigenvector is nonzero only in elements $isuppz(2i-1)$ through $isuppz(2i)$.

Implemented only for $range = 'A'$ or ' I' and $iu-il = n-1$.

$work(1)$

On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

<i>iwork(1)</i>	On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the required minimal size of <i>lwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = <i>i</i> , an internal error has occurred.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `stevr` interface are the following:

<i>d</i>	Holds the vector of length <i>n</i> .
<i>e</i>	Holds the vector of length <i>n</i> .
<i>w</i>	Holds the vector of length <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>), where the values <i>n</i> and <i>m</i> are significant.
<i>isuppz</i>	Holds the vector of length (2*n), where the values (2*m) are significant.
<i>vl</i>	Default value for this element is <i>vl</i> = - <code>HUGE(vl)</code> .
<i>vu</i>	Default value for this element is <i>vu</i> = <code>HUGE(vl)</code> .
<i>il</i>	Default value for this argument is <i>il</i> = 1.
<i>iu</i>	Default value for this argument is <i>iu</i> = <i>n</i> .
<i>abstol</i>	Default value for this element is <i>abstol</i> = 0.0_WP.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted Note that there will be an error condition if <i>ifail</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one of or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present, Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

Normal execution of the routine `?steqr` may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

If it is not clear how much workspace to supply, use a generous value of `lwork` (or `liwork`) for the first run, or set `lwork = -1` (`liwork = -1`).

If `lwork` (or `liwork`) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work, iwork`) on exit. Use this value (`work(1), iwork(1)`) for subsequent runs.

If `lwork = -1` (`liwork = -1`), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work, iwork`). This operation is called a workspace query.

Note that if `lwork` (`liwork`) is less than the minimal required value and is not equal to `-1`, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving nonsymmetric eigenproblems. See also [computational routines](#) that can be called to solve these problems.

[Table "Driver Routines for Solving Nonsymmetric Eigenproblems"](#) lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Nonsymmetric Eigenproblems

Routine Name	Operation performed
<code>gees</code>	Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.
<code>geesx</code>	Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.
<code>geev</code>	Computes the eigenvalues and left and right eigenvectors of a general matrix.
<code>geevx</code>	Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

?gees

Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.

Syntax

FORTRAN 77:

```
call sgees(jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs, work, lwork, bwork,
info)

call dgees(jobvs, sort, select, n, a, lda, sdim, wr, wi, vs, ldvs, work, lwork, bwork,
info)

call cgees(jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs, work, lwork, rwork,
bwork, info)
```

```
call zgees(jobvs, sort, select, n, a, lda, sdim, w, vs, ldvs, work, lwork, rwork,
bwork, info)
```

Fortran 95:

```
call gees(a, wr, wi [,vs] [,select] [,sdim] [,info])
call gees(a, w [,vs] [,select] [,sdim] [,info])
```

C:

```
lapack_int LAPACKE_sgees( int matrix_layout, char jobvs, char sort, LAPACK_S_SELECT2
select, lapack_int n, float* a, lapack_int lda, lapack_int* sdim, float* wr, float*
wi, float* vs, lapack_int ldvs );

lapack_int LAPACKE_dgees( int matrix_layout, char jobvs, char sort, LAPACK_D_SELECT2
select, lapack_int n, double* a, lapack_int lda, lapack_int* sdim, double* wr, double*
wi, double* vs, lapack_int ldvs );

lapack_int LAPACKE_cgees( int matrix_layout, char jobvs, char sort, LAPACK_C_SELECT1
select, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_int* sdim,
lapack_complex_float* w, lapack_complex_float* vs, lapack_int ldvs );

lapack_int LAPACKE_zgees( int matrix_layout, char jobvs, char sort, LAPACK_Z_SELECT1
select, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_int* sdim,
lapack_complex_double* w, lapack_complex_double* vs, lapack_int ldvs );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues, the real Schur form T , and, optionally, the matrix of Schur vectors Z . This gives the Schur factorization $A = Z^* T^* Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left. The leading columns of Z then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \end{bmatrix}$$

where $b*c < 0$. The eigenvalues of such a block are $a \pm \sqrt{bc}$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobvs</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvs</i> = 'N', then Schur vectors are not computed. If <i>jobvs</i> = 'V', then Schur vectors are computed.
<i>sort</i>	CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. If <i>sort</i> = 'N', then eigenvalues are not ordered. If <i>sort</i> = 'S', eigenvalues are ordered (see <i>select</i>).
<i>select</i>	LOGICAL FUNCTION of two REAL arguments for real flavors. LOGICAL FUNCTION of one COMPLEX argument for complex flavors. <i>select</i> must be declared EXTERNAL in the calling subroutine. If <i>sort</i> = 'S', <i>select</i> is used to select eigenvalues to sort to the top left of the Schur form. If <i>sort</i> = 'N', <i>select</i> is not referenced. <i>For real flavors:</i> An eigenvalue $wr(j) + \sqrt{-1} * wi(j)$ is selected if <i>select(wr(j), wi(j))</i> is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected. Note that a selected complex eigenvalue may no longer satisfy <i>select(wr(j), wi(j))= .TRUE.</i> after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case <i>info</i> may be set to <i>n+2</i> (see <i>info</i> below). <i>For complex flavors:</i> An eigenvalue <i>w(j)</i> is selected if <i>select(w(j))</i> is true.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> (<i>n</i> \geq 0).
<i>a, work</i>	REAL for sgees DOUBLE PRECISION for dgees COMPLEX for cgees DOUBLE COMPLEX for zgees. <i>Arrays:</i> <i>a(lda,*)</i> is an array containing the <i>n</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$. INTEGER. The leading dimension of the array <i>a</i> . Must be at least $\max(1, n)$.
<i>ldvs</i>	INTEGER. The leading dimension of the output array <i>vs</i> . Constraints: <i>ldvs</i> ≥ 1 ; <i>ldvs</i> $\geq \max(1, n)$ if <i>jobvs</i> = 'V'.
<i>lwork</i>	INTEGER.

The dimension of the array *work*.

Constraint:

lwork $\geq \max(1, 3n)$ for real flavors;

lwork $\geq \max(1, 2n)$ for complex flavors.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.

rwork

REAL for cgees

DOUBLE PRECISION for zgees

Workspace array, DIMENSION at least $\max(1, n)$. Used in complex flavors only.

bwork

LOGICAL. Workspace array, DIMENSION at least $\max(1, n)$. Not referenced if *sort* = 'N'.

Output Parameters

a

On exit, this array is overwritten by the real-Schur/Schur form *T*.

sdim

INTEGER.

If *sort* = 'N', *sdim* = 0.

If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *select* is true.

Note that for real flavors complex conjugate pairs for which *select* is true for either eigenvalue count as 2.

wr, wi

REAL for sgees

DOUBLE PRECISION for dgees

Arrays, DIMENSION at least $\max(1, n)$ each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form *T*. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

COMPLEX for cgees

DOUBLE COMPLEX for zgees.

Array, DIMENSION at least $\max(1, n)$. Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form *T*.

vs

REAL for sgees

DOUBLE PRECISION for dgees

COMPLEX for cgees

DOUBLE COMPLEX for zgees.

Array *vs*(*ldvs*, *); the second dimension of *vs* must be at least $\max(1, n)$.

If $\text{jobvs} = \text{'V'}$, vs contains the orthogonal/unitary matrix Z of Schur vectors.

If $\text{jobvs} = \text{'N'}$, vs is not referenced.

$\text{work}(1)$
On exit, if $\text{info} = 0$, then $\text{work}(1)$ returns the required minimal size of lwork .

info INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i th parameter had an illegal value.

If $\text{info} = i$, and

$i \leq n$:

the QR algorithm failed to compute all the eigenvalues; elements $1:i\text{lo}-1$ and $i+1:n$ of wr and wi (for real flavors) or w (for complex flavors) contain those eigenvalues which have converged; if $\text{jobvs} = \text{'V'}$, vs contains the matrix which reduces A to its partially converged Schur form;

$i = n+1$:

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

$i = n+2$:

after reordering, round-off changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy $\text{select} = \text{.TRUE.}$. This could also be caused by underflow due to scaling.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gees` interface are the following:

a Holds the matrix A of size (n, n) .

wr Holds the vector of length n . Used in real flavors only.

wi Holds the vector of length n . Used in real flavors only.

w Holds the vector of length n . Used in complex flavors only.

vs Holds the matrix VS of size (n, n) .

jobvs Restored based on the presence of the argument vs as follows:

$\text{jobvs} = \text{'V'}$, if vs is present,

$\text{jobvs} = \text{'N'}$, if vs is omitted.

sort Restored based on the presence of the argument select as follows:

$\text{sort} = \text{'S'}$, if select is present,

$\text{sort} = \text{'N'}$, if select is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geesx

Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.

Syntax

FORTRAN 77:

```
call sgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs, ldvs, rconde,
rcondv, work, lwork, iwork, liwork, bwork, info)

call dgeesx(jobvs, sort, select, sense, n, a, lda, sdim, wr, wi, vs, ldvs, rconde,
rcondv, work, lwork, iwork, liwork, bwork, info)

call cgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs, ldvs, rconde, rcondv,
work, lwork, rwork, bwork, info)

call zgeesx(jobvs, sort, select, sense, n, a, lda, sdim, w, vs, ldvs, rconde, rcondv,
work, lwork, rwork, bwork, info)
```

Fortran 95:

```
call geesx(a, wr, wi [,vs] [,select] [,sdim] [,rconde] [,rcondv] [,info])
call geesx(a, w [,vs] [,select] [,sdim] [,rconde] [,rcondv] [,info])
```

C:

```
lapack_int LAPACKE_sgeesx( int matrix_layout, char jobvs, char sort, LAPACK_S_SELECT2
select, char sense, lapack_int n, float* a, lapack_int lda, lapack_int* sdim, float*
wr, float* wi, float* vs, lapack_int ldvs, float* rconde, float* rcondv );

lapack_int LAPACKE_dgeesx( int matrix_layout, char jobvs, char sort, LAPACK_D_SELECT2
select, char sense, lapack_int n, double* a, lapack_int lda, lapack_int* sdim, double*
wr, double* wi, double* vs, lapack_int ldvs, double* rconde, double* rcondv );

lapack_int LAPACKE_cgeesx( int matrix_layout, char jobvs, char sort, LAPACK_C_SELECT1
select, char sense, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_int*
sdim, lapack_complex_float* w, lapack_complex_float* vs, lapack_int ldvs, float*
rconde, float* rcondv );

lapack_int LAPACKE_zgeesx( int matrix_layout, char jobvs, char sort, LAPACK_Z_SELECT1
select, char sense, lapack_int n, lapack_complex_double* a, lapack_int lda,
lapack_int* sdim, lapack_complex_double* w, lapack_complex_double* vs, lapack_int ldvs,
double* rconde, double* rcondv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues, the real-Schur/Schur form T , and, optionally, the matrix of Schur vectors Z . This gives the Schur factorization $A = Z^* T^* Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (*rconde*); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (*rcondv*). The leading columns of Z form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers *rconde* and *rcondv*, see [LUG], Section 4.10 (where these quantities are called *s* and *sep* respectively).

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$\begin{bmatrix} & \cdots \\ & \vdots \\ & \cdots \end{bmatrix}$$

where $b*c < 0$. The eigenvalues of such a block are

$$\lambda_1, \lambda_2, \dots, \lambda_n$$

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobvs</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvs</i> = 'N', then Schur vectors are not computed. If <i>jobvs</i> = 'V', then Schur vectors are computed.
<i>sort</i>	CHARACTER*1. Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form. If <i>sort</i> = 'N', then eigenvalues are not ordered. If <i>sort</i> = 'S', eigenvalues are ordered (see <i>select</i>).
<i>select</i>	LOGICAL FUNCTION of two REAL arguments for real flavors. LOGICAL FUNCTION of one COMPLEX argument for complex flavors. <i>select</i> must be declared EXTERNAL in the calling subroutine. If <i>sort</i> = 'S', <i>select</i> is used to select eigenvalues to sort to the top left of the Schur form.

If $sort = 'N'$, $select$ is not referenced.

For real flavors:

An eigenvalue $wr(j) + \sqrt{-1} * wi(j)$ is selected if $select(wr(j), wi(j))$ is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that a selected complex eigenvalue may no longer satisfy $select(wr(j), wi(j)) = .TRUE.$ after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case $info$ may be set to $n+2$ (see $info$ below).

For complex flavors:

An eigenvalue $w(j)$ is selected if $select(w(j))$ is true.

sense CHARACTER*1. Must be ' N ', ' E ', ' V ', or ' B '. Determines which reciprocal condition number are computed.

If $sense = 'N'$, none are computed;

If $sense = 'E'$, computed for average of selected eigenvalues only;

If $sense = 'V'$, computed for selected right invariant subspace only;

If $sense = 'B'$, computed for both.

If $sense$ is ' E ', ' V ', or ' B ', then $sort$ must equal ' S '.

n INTEGER. The order of the matrix A ($n \geq 0$).

a, work REAL for sgeesx

DOUBLE PRECISION for dgeesx

COMPLEX for cgeesx

DOUBLE COMPLEX for zgeesx.

Arrays:

a($lda, *$) is an array containing the n -by- n matrix A .

The second dimension of *a* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of the array *a*. Must be at least $\max(1, n)$.

ldvs INTEGER. The leading dimension of the output array *vs*. Constraints:

$ldvs \geq 1$;

$ldvs \geq \max(1, n)$ if $jobvs = 'V'$.

lwork INTEGER.

The dimension of the array *work*. Constraint:

$lwork \geq \max(1, 3n)$ for real flavors;

$lwork \geq \max(1, 2n)$ for complex flavors.

Also, if $sense = 'E'$, ' V ', or ' B ', then

$lwork \geq n+2*sdim*(n-sdim)$ for real flavors;

$lwork \geq 2*sdim*(n-sdim)$ for complex flavors;

where $sdim$ is the number of selected eigenvalues computed by this routine.

Note that $2*sdim*(n-sdim) \leq n*n/2$. Note also that an error is only returned if $lwork < \max(1, 2*n)$, but if $sense = 'E'$, or ' ∇ ', or ' B ' this may not be large enough.

For good performance, $lwork$ must generally be larger.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates upper bound on the optimal size of the array $work$, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by xerbla.

$iwork$

INTEGER.

Workspace array, DIMENSION ($liwork$). Used in real flavors only. Not referenced if $sense = 'N'$ or ' E '.

$liwork$

INTEGER.

The dimension of the array $iwork$. Used in real flavors only.

Constraint:

$liwork \geq 1$;

if $sense = 'V'$ or ' B ', $liwork \geq sdim*(n-sdim)$.

$rwork$

REAL for cgeesx

DOUBLE PRECISION for zgeesx

Workspace array, DIMENSION at least $\max(1, n)$. Used in complex flavors only.

$bwork$

LOGICAL. Workspace array, DIMENSION at least $\max(1, n)$. Not referenced if $sort = 'N'$.

Output Parameters

a

On exit, this array is overwritten by the real-Schur/Schur form T .

$sdim$

INTEGER.

If $sort = 'N'$, $sdim = 0$.

If $sort = 'S'$, $sdim$ is equal to the number of eigenvalues (after sorting) for which $select$ is true.

Note that for real flavors complex conjugate pairs for which $select$ is true for either eigenvalue count as 2.

wr, wi

REAL for sgeesx

DOUBLE PRECISION for dgeesx

Arrays, DIMENSION at least max (1, n) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form T . Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

*w*COMPLEX for `cgeesx`DOUBLE COMPLEX for `zgeesx`.

Array, DIMENSION at least max(1, n). Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form T .

*vs*REAL for `sgeesx`DOUBLE PRECISION for `dgeesx`COMPLEX for `cgeesx`DOUBLE COMPLEX for `zgeesx`.

Array $vs(ldvs,*)$; the second dimension of vs must be at least max(1, n).

If $jobvs = 'V'$, vs contains the orthogonal/unitary matrix Z of Schur vectors.

If $jobvs = 'N'$, vs is not referenced.

rconde, rcondv

REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

If $sense = 'E'$ or ' B ', $rconde$ contains the reciprocal condition number for the average of the selected eigenvalues.

If $sense = 'N'$ or ' V ', $rconde$ is not referenced.

If $sense = 'V'$ or ' B ', $rcondv$ contains the reciprocal condition number for the selected right invariant subspace.

If $sense = 'N'$ or ' E ', $rcondv$ is not referenced.

work(1)

On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i th parameter had an illegal value.

If $info = i$, and

$i \leq n$:

the QR algorithm failed to compute all the eigenvalues; elements 1: $ilo-1$ and $i+1:n$ of wr and wi (for real flavors) or w (for complex flavors) contain those eigenvalues which have converged; if $jobvs = 'V'$, vs contains the transformation which reduces A to its partially converged Schur form;

$i = n+1$:

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

i = *n*+2:

after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy *select* = .TRUE.. This could also be caused by underflow due to scaling.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geesx` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>wr</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>wi</i>	Holds the vector of length (<i>n</i>). Used in real flavors only.
<i>w</i>	Holds the vector of length (<i>n</i>). Used in complex flavors only.
<i>vs</i>	Holds the matrix <i>VS</i> of size (<i>n</i> , <i>n</i>).
<i>jobvs</i>	Restored based on the presence of the argument <i>vs</i> as follows: <i>jobvs</i> = 'V', if <i>vs</i> is present, <i>jobvs</i> = 'N', if <i>vs</i> is omitted.
<i>sort</i>	Restored based on the presence of the argument <i>select</i> as follows: <i>sort</i> = 'S', if <i>select</i> is present, <i>sort</i> = 'N', if <i>select</i> is omitted.
<i>sense</i>	Restored based on the presence of arguments <i>rconde</i> and <i>rcondv</i> as follows: <i>sense</i> = 'B', if both <i>rconde</i> and <i>rcondv</i> are present, <i>sense</i> = 'E', if <i>rconde</i> is present and <i>rcondv</i> omitted, <i>sense</i> = 'V', if <i>rconde</i> is omitted and <i>rcondv</i> present, <i>sense</i> = 'N', if both <i>rconde</i> and <i>rcondv</i> are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?geev

Computes the eigenvalues and left and right eigenvectors of a general matrix.

Syntax**FORTRAN 77:**

```
call sgeev(jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork, info)
call dgeev(jobvl, jobvr, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, work, lwork, info)
call cgeev(jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
call zgeev(jobvl, jobvr, n, a, lda, w, vl, ldvl, vr, ldvr, work, lwork, rwork, info)
```

Fortran 95:

```
call geev(a, wr, wi [,vl] [,vr] [,info])
call geev(a, w [,vl] [,vr] [,info])
```

C:

```
lapack_int LAPACKE_sgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
float* a, lapack_int lda, float* wr, float* wi, float* vl, lapack_int ldvl, float* vr,
lapack_int ldvr );
lapack_int LAPACKE_dgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
double* a, lapack_int lda, double* wr, double* wi, double* vl, lapack_int ldvl,
double* vr, lapack_int ldvr );
lapack_int LAPACKE_cgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* w, lapack_complex_float*
vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr );
lapack_int LAPACKE_zgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* w,
lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int
ldvr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues and, optionally, the left and/or right eigenvectors. The right eigenvector $v(j)$ of A satisfies

$$A^*v(j) = \lambda(j)^*v(j)$$

where $\lambda(j)$ is its eigenvalue.

The left eigenvector $u(j)$ of A satisfies

$$u(j)^H * A = \lambda(j)^* u(j)^H$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$. The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>jobvl</i>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>V</code> '. If <i>jobvl</i> = ' <code>N</code> ', then left eigenvectors of <i>A</i> are not computed. If <i>jobvl</i> = ' <code>V</code> ', then left eigenvectors of <i>A</i> are computed.
<i>jobvr</i>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>V</code> '. If <i>jobvr</i> = ' <code>N</code> ', then right eigenvectors of <i>A</i> are not computed. If <i>jobvr</i> = ' <code>V</code> ', then right eigenvectors of <i>A</i> are computed.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> ($n \geq 0$).
<i>a, work</i>	REAL for <code>sgeev</code> DOUBLE PRECISION for <code>dgeev</code> COMPLEX for <code>cgeev</code> DOUBLE COMPLEX for <code>zgeev</code> . Arrays: <i>a</i> (<i>lda</i> ,*) is an array containing the <i>n</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least <code>max(1, n)</code> . <i>work</i> is a workspace array, its dimension <code>max(1, lwork)</code> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . Must be at least <code>max(1, n)</code> .
<i>ldvl, ldvr</i>	INTEGER. The leading dimensions of the output arrays <i>vl</i> and <i>vr</i> , respectively. Constraints: $ldvl \geq 1; ldvr \geq 1$. If <i>jobvl</i> = ' <code>V</code> ', <i>ldvl</i> $\geq \max(1, n)$; If <i>jobvr</i> = ' <code>V</code> ', <i>ldvr</i> $\geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . Constraint for real flavors: $lwork \geq \max(1, 3n)$. If computing eigenvectors (<i>jobvl</i> = ' <code>V</code> ' or <i>jobvr</i> = ' <code>V</code> '), <i>lwork</i> $\geq \max(1, 4n)$. Constraint for complex flavors: $lwork \geq \max(1, 2n)$.

For good performance, $lwork$ must generally be larger.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`.

$rwork$

REAL for `cgeev`

DOUBLE PRECISION for `zgeev`

Workspace array, size at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

a

On exit, this array is overwritten by intermediate results.

wr, wi

REAL for `sgeev`

DOUBLE PRECISION for `dgeev`

Arrays, size at least $\max(1, n)$ each.

Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

COMPLEX for `cgeev`

DOUBLE COMPLEX for `zgeev`.

Array, size at least $\max(1, n)$.

Contains the computed eigenvalues.

vl, vr

REAL for `sgeev`

DOUBLE PRECISION for `dgeev`

COMPLEX for `cgeev`

DOUBLE COMPLEX for `zgeev`.

Arrays:

$vl(ldbl, *)$; the second dimension of vl must be at least $\max(1, n)$.

If $jobvl = 'V'$, the left eigenvectors $u(j)$ are stored one after another in the columns of vl , in the same order as their eigenvalues.

If $jobvl = 'N'$, vl is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $u(j) = vl(:, j)$, the j -th column of vl .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $u(j) = vl(:, j) + i * vl(:, j+1)$ and $u(j+1) = vl(:, j) - i * vl(:, j+1)$, where $i = \sqrt{-1}$.

For complex flavors:

$u(j) = vl(:, j)$, the j -th column of vl .

$vr(livr, *)$; the second dimension of vr must be at least $\max(1, n)$.

If $\text{jobvr} = \text{'V'}$, the right eigenvectors $v(j)$ are stored one after another in the columns of vr , in the same order as their eigenvalues.

If $\text{jobvr} = \text{'N'}$, vr is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $v(j) = vr(:, j)$, the j -th column of vr .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $v(j) = vr(:, j) + i * vr(:, j+1)$ and $v(j+1) = vr(:, j) - i * vr(:, j+1)$, where $i = \sqrt{-1}$.

For complex flavors:

$v(j) = vr(:, j)$, the j -th column of vr .

$\text{work}(1)$

On exit, if $\text{info} = 0$, then $\text{work}(1)$ returns the required minimal size of lwork .

info

INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i th parameter had an illegal value.

If $\text{info} = i$, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; elements $i+1:n$ of wr and wi (for real flavors) or w (for complex flavors) contain those eigenvalues which have converged.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geev` interface are the following:

a Holds the matrix A of size (n, n) .

wr Holds the vector of length n . Used in real flavors only.

wi Holds the vector of length n . Used in real flavors only.

w Holds the vector of length n . Used in complex flavors only.

vl Holds the matrix VL of size (n, n) .

vr Holds the matrix VR of size (n, n) .

jobvl Restored based on the presence of the argument vl as follows:

$\text{jobvl} = \text{'V'}$, if vl is present,

$\text{jobvl} = \text{'N'}$, if vl is omitted.

jobvr Restored based on the presence of the argument vr as follows:

$\text{jobvr} = \text{'V'}$, if vr is present,

$\text{jobvr} = \text{'N'}$, if vr is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine exits immediately with an error and does not provide any information on the recommended workspace.

?geevx

Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

Syntax

FORTRAN 77:

```
call sggevx(balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, iwork, info)
call dgeevx(balanc, jobvl, jobvr, sense, n, a, lda, wr, wi, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, iwork, info)
call cgeevx(balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, rwork, info)
call zgeevx(balanc, jobvl, jobvr, sense, n, a, lda, w, vl, ldvl, vr, ldvr, ilo, ihi, scale, abnrm, rconde, rcondv, work, lwork, rwork, info)
```

Fortran 95:

```
call ggeevx(a, wr, wi [, vl] [, vr] [, balanc] [, ilo] [, ihi] [, scale] [, abnrm] [, rconde] [, rcondv] [, info])
call ggeevx(a, w [, vl] [, vr] [, balanc] [, ilo] [, ihi] [, scale] [, abnrm] [, rconde] [, rcondv] [, info])
```

C:

```
lapack_int LAPACKE_sggevx( int matrix_layout, char balanc, char jobvl, char jobvr, char sense, lapack_int n, float* a, lapack_int lda, float* wr, float* wi, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, float* scale, float* abnrm, float* rconde, float* rcondv);
lapack_int LAPACKE_dgeevx( int matrix_layout, char balanc, char jobvl, char jobvr, char sense, lapack_int n, double* a, lapack_int lda, double* wr, double* wi, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, double* scale, double* abnrm, double* rconde, double* rcondv);
lapack_int LAPACKE_cgeevx( int matrix_layout, char balanc, char jobvl, char jobvr, char sense, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* w, lapack_complex_float* vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, float* scale, float* abnrm, float* rconde, float* rcondv);
```

```
lapack_int LAPACKE_zgeevx( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double*
w, lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int
ldvr, lapack_int* ilo, lapack_int* ihi, double* scale, double* abnrm, double* rconde,
double* rcondv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes for an n -by- n real/complex nonsymmetric matrix A , the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ilo , ihi , $scale$, and $abnrm$), reciprocal condition numbers for the eigenvalues ($rconde$), and reciprocal condition numbers for the right eigenvectors ($rcondv$).

The right eigenvector $v(j)$ of A satisfies

$$A^*v(j) = \lambda(j)^*v(j)$$

where $\lambda(j)$ is its eigenvalue.

The left eigenvector $u(j)$ of A satisfies

$$u(j)^H A = \lambda(j)^* u(j)^H$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$. The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation $D^*A^*\text{inv}(D)$, where D is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see [LUG], Section 4.10.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>balanc</i>	CHARACTER*1. Must be 'N', 'P', 'S', or 'B'. Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues. If <i>balanc</i> = 'N', do not diagonally scale or permute; If <i>balanc</i> = 'P', perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale; If <i>balanc</i> = 'S', diagonally scale the matrix, i.e. replace A by $D^*A^*\text{inv}(D)$, where D is a diagonal matrix chosen to make the rows and columns of A more equal in norm. Do not permute; If <i>balanc</i> = 'B', both diagonally scale and permute A .
---------------	--

Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

jobvl

CHARACTER*1. Must be 'N' or 'V'.

If *jobvl* = 'N', left eigenvectors of *A* are not computed;

If *jobvl* = 'V', left eigenvectors of *A* are computed.

If *sense* = 'E' or 'B', then *jobvl* must be 'V'.

jobvr

CHARACTER*1. Must be 'N' or 'V'.

If *jobvr* = 'N', right eigenvectors of *A* are not computed;

If *jobvr* = 'V', right eigenvectors of *A* are computed.

If *sense* = 'E' or 'B', then *jobvr* must be 'V'.

sense

CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.

If *sense* = 'N', none are computed;

If *sense* = 'E', computed for eigenvalues only;

If *sense* = 'V', computed for right eigenvectors only;

If *sense* = 'B', computed for eigenvalues and right eigenvectors.

If *sense* is 'E' or 'B', both left and right eigenvectors must also be computed (*jobvl* = 'V' and *jobvr* = 'V').

n

INTEGER. The order of the matrix *A* (*n* \geq 0).

a, work

REAL for sgeevx

DOUBLE PRECISION for dgeevx

COMPLEX for cgeevx

DOUBLE COMPLEX for zgeevx.

Arrays:

a(*lda*,*) is an array containing the *n*-by-*n* matrix *A*.

The second dimension of *a* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The leading dimension of the array *a*. Must be at least $\max(1, n)$.

ldvl, ldvr

INTEGER. The leading dimensions of the output arrays *vl* and *vr*, respectively.

Constraints:

ldvl \geq 1; *ldvr* \geq 1.

If *jobvl* = 'V', *ldvl* $\geq \max(1, n)$;

If *jobvr* = 'V', *ldvr* $\geq \max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

For real flavors:

If *sense* = 'N' or 'E', *lwork* $\geq \max(1, 2n)$, and if *jobvl* = 'V' or *jobvr* = 'V', *lwork* $\geq 3n$;

If *sense* = 'V' or 'B', *lwork* $\geq n*(n+6)$.

For good performance, *lwork* must generally be larger.

For complex flavors:

If *sense* = 'N' or 'E', *lwork* $\geq \max(1, 2n)$;

If *sense* = 'V' or 'B', *lwork* $\geq n^2 + 2n$. For good performance, *lwork* must generally be larger.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [xerbla](#).

rwork

REAL for `cgeevx`

DOUBLE PRECISION for `zgeevx`

Workspace array, DIMENSION at least $\max(1, 2n)$. Used in complex flavors only.

iwork

INTEGER.

Workspace array, DIMENSION at least $\max(1, 2n-2)$. Used in real flavors only. Not referenced if *sense* = 'N' or 'E'.

Output Parameters

a

On exit, this array is overwritten.

If *jobvl* = 'V' or *jobvr* = 'V', it contains the real-Schur/Schur form of the balanced version of the input matrix *A*.

wr, wi

REAL for `sgeevx`

DOUBLE PRECISION for `dgeevx`

Arrays, DIMENSION at least max (1, *n*) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w

COMPLEX for `cgeevx`

DOUBLE COMPLEX for `zgeevx`.

Array, DIMENSION at least $\max(1, n)$. Contains the computed eigenvalues.

vl, vr

REAL for `sgeevx`

DOUBLE PRECISION for `dgeevx`

COMPLEX for `cgeevx`

DOUBLE COMPLEX for `zgeevx`.

Arrays:

$vl(Idvl,*)$; the second dimension of vl must be at least $\max(1, n)$.

If $jobvl = 'V'$, the left eigenvectors $u(j)$ are stored one after another in the columns of vl , in the same order as their eigenvalues.

If $jobvl = 'N'$, vl is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $u(j) = vl(:, j)$, the j -th column of vl .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $u(j) = vl(:, j) + i * vl(:, j+1)$ and $u(j+1) = vl(:, j) - i * vl(:, j+1)$, where $i = \sqrt{-1}$.

For complex flavors:

$u(j) = vl(:, j)$, the j -th column of vl .

$vr(Idvr,*)$; the second dimension of vr must be at least $\max(1, n)$.

If $jobvr = 'V'$, the right eigenvectors $v(j)$ are stored one after another in the columns of vr , in the same order as their eigenvalues.

If $jobvr = 'N'$, vr is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $v(j) = vr(:, j)$, the j -th column of vr .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $v(j) = vr(:, j) + i * vr(:, j+1)$ and $v(j+1) = vr(:, j) - i * vr(:, j+1)$, where $i = \sqrt{-1}$.

For complex flavors:

$v(j) = vr(:, j)$, the j -th column of vr .

ilo, ihi

INTEGER. ilo and ihi are integer values determined when A was balanced.

The balanced $A(i, j) = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$.

If $balanc = 'N'$ or $'S'$, $ilo = 1$ and $ihi = n$.

$scale$

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, DIMENSION at least $\max(1, n)$. Details of the permutations and scaling factors applied when balancing A .

If $P(j)$ is the index of the row and column interchanged with row and column j , and $D(j)$ is the scaling factor applied to row and column j , then

$$\begin{aligned} scale(j) &= P(j), \text{ for } j = 1, \dots, ilo-1 \\ &= D(j), \text{ for } j = ilo, \dots, ihi \\ &= P(j) \text{ for } j = ihi+1, \dots, n. \end{aligned}$$

The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

$abnrm$

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

rconde, rcondv REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION at least max(1, *n*) each.

rconde(j) is the reciprocal condition number of the *j*-th eigenvalue.

rcondv(j) is the reciprocal condition number of the *j*-th right eigenvector.

work(1) On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors or condition numbers have been computed; elements 1: *i/o*-1 and *i+1:n* of *wr* and *wi* (for real flavors) or *w* (for complex flavors) contain eigenvalues which have converged.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `geevx` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>wr</i>	Holds the vector of length <i>n</i> . Used in real flavors only.
<i>wi</i>	Holds the vector of length <i>n</i> . Used in real flavors only.
<i>w</i>	Holds the vector of length <i>n</i> . Used in complex flavors only.
<i>vl</i>	Holds the matrix <i>VL</i> of size (<i>n</i> , <i>n</i>).
<i>vr</i>	Holds the matrix <i>VR</i> of size (<i>n</i> , <i>n</i>).
<i>scale</i>	Holds the vector of length <i>n</i> .
<i>rconde</i>	Holds the vector of length <i>n</i> .
<i>rcondv</i>	Holds the vector of length <i>n</i> .
<i>balanc</i>	Must be 'N', 'B', 'P' or 'S'. The default value is 'N'.
<i>jobvl</i>	Restored based on the presence of the argument <i>vl</i> as follows: <i>jobvl</i> = 'V', if <i>vl</i> is present, <i>jobvl</i> = 'N', if <i>vl</i> is omitted.
<i>jobvr</i>	Restored based on the presence of the argument <i>vr</i> as follows:

	<i>jobvr</i> = 'V', if <i>vr</i> is present, <i>jobvr</i> = 'N', if <i>vr</i> is omitted.
<i>sense</i>	Restored based on the presence of arguments <i>rconde</i> and <i>rcondv</i> as follows: <i>sense</i> = 'B', if both <i>rconde</i> and <i>rcondv</i> are present, <i>sense</i> = 'E', if <i>rconde</i> is present and <i>rcondv</i> omitted, <i>sense</i> = 'V', if <i>rconde</i> is omitted and <i>rcondv</i> present, <i>sense</i> = 'N', if both <i>rconde</i> and <i>rcondv</i> are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

Singular Value Decomposition - LAPACK Driver Routines

[Table "Driver Routines for Singular Value Decomposition"](#) lists the LAPACK driver routines that perform singular value decomposition for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are the same except that the first character is removed.

Driver Routines for Singular Value Decomposition

Routine Name	Operation performed
?gesvd	Computes the singular value decomposition of a general rectangular matrix.
?gesdd	Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.
?gejsv	Computes the singular value decomposition of a real matrix using a preconditioned Jacobi SVD method.
?gesvj	Computes the singular value decomposition of a real matrix using Jacobi plane rotations.
?ggsvd	Computes the generalized singular value decomposition of a pair of general rectangular matrices.

Singular Value Decomposition - LAPACK Computational Routines

?gesvd

Computes the singular value decomposition of a general rectangular matrix.

Syntax

FORTRAN 77:

```
call sgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info)
```

```
call dgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, info)
call cgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info)
call zgesvd(jobu, jobvt, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, info)
```

Fortran 95:

```
call gesvd(a, s [,u] [,vt] [,ww] [,job] [,info])
```

C:

```
lapack_int LAPACKE_sgesvd( int matrix_layout, char jobu, char jobvt, lapack_int m,
                           lapack_int n, float* a, lapack_int lda, float* s, float* u, lapack_int ldu, float* vt,
                           lapack_int ldvt, float* superb );

lapack_int LAPACKE_dgesvd( int matrix_layout, char jobu, char jobvt, lapack_int m,
                           lapack_int n, double* a, lapack_int lda, double* s, double* u, lapack_int ldu, double* vt,
                           lapack_int ldvt, double* superb );

lapack_int LAPACKE_cgesvd( int matrix_layout, char jobu, char jobvt, lapack_int m,
                           lapack_int n, lapack_complex_float* a, lapack_int lda, float* s, lapack_complex_float* u,
                           lapack_int ldu, lapack_complex_float* vt, lapack_int ldvt, float* superb );

lapack_int LAPACKE_zgesvd( int matrix_layout, char jobu, char jobvt, lapack_int m,
                           lapack_int n, lapack_complex_double* a, lapack_int lda, double* s,
                           lapack_complex_double* u, lapack_int ldu, lapack_complex_double* vt, lapack_int ldvt,
                           double* superb );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the singular value decomposition (SVD) of a real/complex m -by- n matrix A , optionally computing the left and/or right singular vectors. The SVD is written as

$A = U^* \Sigma V^T$ for real routines

$A = U^* \Sigma V^H$ for complex routines

where Σ is an m -by- n matrix which is zero except for its $\min(m,n)$ diagonal elements, U is an m -by- m orthogonal/unitary matrix, and V is an n -by- n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A ; they are real and non-negative, and are returned in descending order. The first $\min(m,n)$ columns of U and V are the left and right singular vectors of A .

Note that the routine returns V^T (for real flavors) or V^H (for complex flavors), not V .

Input Parameters

The data types are given for the Fortran interface, except for *superb*. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

jobu CHARACTER*1. Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix U .

If *jobu* = 'A', all m columns of U are returned in the array *u*;

if *jobu* = 'S', the first $\min(m, n)$ columns of U (the left singular vectors) are returned in the array *u*;

if $\text{jobu} = \text{'O'}$, the first $\min(m, n)$ columns of U (the left singular vectors) are overwritten on the array a ;

if $\text{jobu} = \text{'N'}$, no columns of U (no left singular vectors) are computed.

jobvt

CHARACTER*1. Must be ' A ', ' S ', ' O ', or ' N '. Specifies options for computing all or part of the matrix V^T/V^H .

If $\text{jobvt} = \text{'A'}$, all n rows of V^T/V^H are returned in the array vt ;

if $\text{jobvt} = \text{'S'}$, the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors) are returned in the array vt ;

if $\text{jobvt} = \text{'O'}$, the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors) are overwritten on the array a ;

if $\text{jobvt} = \text{'N'}$, no rows of V^T/V^H (no right singular vectors) are computed.

jobvt and jobu cannot both be ' O '.

m

INTEGER. The number of rows of the matrix A ($m \geq 0$).

n

INTEGER. The number of columns in A ($n \geq 0$).

a, work

REAL for sgesvd

DOUBLE PRECISION for dgesvd

COMPLEX for cgesvd

DOUBLE COMPLEX for zgesvd.

Arrays:

$a(\text{lda},*)$ is an array containing the m -by- n matrix A .

The second dimension of a must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The leading dimension of the array a .

Must be at least $\max(1, m)$.

ldu, ldvt

INTEGER. The leading dimensions of the output arrays u and vt , respectively.

Constraints:

$\text{ldu} \geq 1; \text{ldvt} \geq 1$.

If $\text{jobu} = \text{'S'}$ or ' A ', $\text{ldu} \geq m$;

If $\text{jobvt} = \text{'A'}$, $\text{ldvt} \geq n$;

If $\text{jobvt} = \text{'S'}$, $\text{ldvt} \geq \min(m, n)$.

lwork

INTEGER.

The dimension of the array work .

Constraints:

$\text{lwork} \geq 1$

$\text{lwork} \geq \max(3 * \min(m, n) + \max(m, n), 5 * \min(m, n))$ (for real flavors);

$lwork \geq 2*\min(m, n) + \max(m, n)$ (for complex flavors).

For good performance, $lwork$ must generally be larger.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by `xerbla`. See *Application Notes* for details.

$rwork$

REAL for `cgesvd`

DOUBLE PRECISION for `zgesvd`

Workspace array, DIMENSION at least $\max(1, 5*\min(m, n))$. Used in complex flavors only.

Output Parameters

a

On exit,

If $jobu = 'O'$, a is overwritten with the first $\min(m, n)$ columns of U (the left singular vectors stored columnwise);

If $jobvt = 'O'$, a is overwritten with the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors stored rowwise);

If $jobu \neq 'O'$ and $jobvt \neq 'O'$, the contents of a are destroyed.

s

REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Array, DIMENSION at least $\max(1, \min(m, n))$. Contains the singular values of A sorted so that $s(i) \geq s(i+1)$.

u, vt

REAL for `sgesvd`

DOUBLE PRECISION for `dgesvd`

COMPLEX for `cgesvd`

DOUBLE COMPLEX for `zgesvd`.

Arrays:

$u(lu, *)$; the second dimension of u must be at least $\max(1, m)$ if $jobu = 'A'$, and at least $\max(1, \min(m, n))$ if $jobu = 'S'$.

If $jobu = 'A'$, u contains the m -by- m orthogonal/unitary matrix U .

If $jobu = 'S'$, u contains the first $\min(m, n)$ columns of U (the left singular vectors stored column-wise).

If $jobu = 'N'$ or $'O'$, u is not referenced.

$vt(lvvt, *)$; the second dimension of vt must be at least $\max(1, n)$.

If $jobvt = 'A'$, vt contains the n -by- n orthogonal/unitary matrix V^T/V^H .

If $jobvt = 'S'$, vt contains the first $\min(m, n)$ rows of V^T/V^H (the right singular vectors stored row-wise).

If $jobvt = 'N'$ or $'O'$, vt is not referenced.

$work$

On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

For real flavors:

If $\text{info} > 0$, $\text{work}(2:\min(m,n))$ contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in s (not necessarily sorted). B satisfies $A = u^* B^* v t$, so it has the same singular values as A , and singular vectors related by u and $v t$.

$rwork$

On exit (for complex flavors), if $\text{info} > 0$, $rwork(1:\min(m,n)-1)$ contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in s (not necessarily sorted). B satisfies $A = u^* B^* v t$, so it has the same singular values as A , and singular vectors related by u and $v t$.

$superb$ (C interface)

On exit, $superb(0:\min(m,n)-2)$ contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in s (not necessarily sorted). B satisfies $A = u^* B^* v t$, so it has the same singular values as A , and singular vectors related by u and $v t$.

$info$

INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th parameter had an illegal value.

If $\text{info} = i$, then if ?bdsqr did not converge, i specifies how many superdiagonals of the intermediate bidiagonal form B did not converge to zero.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gesvd` interface are the following:

a

Holds the matrix A of size (m, n) .

s

Holds the vector of length $\min(m, n)$.

u

If present and is a square m -by- m matrix, on exit contains the m -by- m orthogonal/unitary matrix U .

Otherwise, if present, on exit contains the first $\min(m, n)$ columns of the matrix U (left singular vectors stored column-wise).

vt

If present and is a square n -by- n matrix, on exit contains the n -by- n orthogonal/unitary matrix V^T/V^H .

Otherwise, if present, on exit contains the first $\min(m, n)$ rows of the matrix V^T/V^H (right singular vectors stored row-wise).

ww

Holds the vector of length $\min(m, n)-1$. ww contains the unconverged superdiagonal elements of an upper bidiagonal matrix B whose diagonal is in s (not necessarily sorted). B satisfies $A = U^* B^* V T$, so it has the same singular values as A , and singular vectors related by U and $V T$.

job

Must be either '`N`', or '`U`', or '`V`'. The default value is '`N`'.

If $\text{job} = \text{'U'}$, and u is not present, then u is returned in the array a .

If $job = 'V'$, and vt is not present, then vt is returned in the array a .

Application Notes

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array $work$ on exit. Use this value ($work(1)$) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gesdd

Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.

Syntax

FORTRAN 77:

```
call sgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
call dgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, iwork, info)
call cgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info)
call zgesdd(jobz, m, n, a, lda, s, u, ldu, vt, ldvt, work, lwork, rwork, iwork, info)
```

Fortran 95:

```
call gesdd(a, s [,u] [,vt] [,jobz] [,info])
```

C:

```
lapack_int LAPACKE_sgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
float* a, lapack_int lda, float* s, float* u, lapack_int ldu, float* vt, lapack_int ldvt );

lapack_int LAPACKE_dgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
double* a, lapack_int lda, double* s, double* u, lapack_int ldu, double* vt,
lapack_int ldvt );

lapack_int LAPACKE_cgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, float* s, lapack_complex_float* u, lapack_int ldu,
lapack_complex_float* vt, lapack_int ldvt );

lapack_int LAPACKE_zgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, double* s, lapack_complex_double* u,
lapack_int ldu, lapack_complex_double* vt, lapack_int ldvt );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the singular value decomposition (SVD) of a real/complex m -by- n matrix A , optionally computing the left and/or right singular vectors.

If singular vectors are desired, it uses a divide-and-conquer algorithm. The SVD is written

$A = U \Sigma V'$ for real routines,

$A = U \Sigma \text{conjg}(V')$ for complex routines,

where Σ is an m -by- n matrix which is zero except for its $\min(m,n)$ diagonal elements, U is an m -by- m orthogonal/unitary matrix, and V is an n -by- n orthogonal/unitary matrix. The diagonal elements of Σ are the singular values of A ; they are real and non-negative, and are returned in descending order. The first $\min(m,n)$ columns of U and V are the left and right singular vectors of A .

Note that the routine returns $vt = V'$ (for real flavors) or $vt = \text{conjg}(V')$ (for complex flavors), not V .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>jobz</i>	<p>CHARACTER*1. Must be '<code>A</code>', '<code>S</code>', '<code>O</code>', or '<code>N</code>'.</p> <p>Specifies options for computing all or part of the matrix U.</p> <p>If $jobz = 'A'$, all m columns of U and all n rows of $V'/\text{conjg}(V)$ are returned in the arrays u and vt;</p> <p>if $jobz = 'S'$, the first $\min(m, n)$ columns of U and the first $\min(m, n)$ rows of $V'/\text{conjg}(V)$ are returned in the arrays u and vt;</p> <p>if $jobz = 'O'$, then</p> <p>if $m \geq n$, the first n columns of U are overwritten in the array a and all rows of $V'/\text{conjg}(V)$ are returned in the array vt;</p> <p>if $m < n$, all columns of U are returned in the array u and the first m rows of $V'/\text{conjg}(V)$ are overwritten in the array a;</p> <p>if $jobz = 'N'$, no columns of U or rows of $V'/\text{conjg}(V)$ are computed.</p>
<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in A ($n \geq 0$).
<i>a, work</i>	<p>REAL for <code>s gesdd</code></p> <p>DOUBLE PRECISION for <code>d gesdd</code></p> <p>COMPLEX for <code>c gesdd</code></p> <p>DOUBLE COMPLEX for <code>z gesdd</code>.</p> <p>Arrays: $a(lda,*)$ is an array containing the m-by-n matrix A. The second dimension of a must be at least $\max(1, n)$. $work$ is a workspace array, its dimension $\max(1, lwork)$.</p>

<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . Must be at least $\max(1, m)$.
<i>ldu</i> , <i>ldvt</i>	INTEGER. The leading dimensions of the output arrays <i>u</i> and <i>vt</i> , respectively.
	Constraints:
	<i>ldu</i> ≥ 1 ; <i>ldvt</i> ≥ 1 .
	If <i>jobz</i> = 'S' or 'A', or <i>jobz</i> = 'O' and $m < n$, then <i>ldu</i> $\geq m$;
	If <i>jobz</i> = 'A' or <i>jobz</i> = 'O' and $m \geq n$, then <i>ldvt</i> $\geq n$;
	If <i>jobz</i> = 'S', <i>ldvt</i> $\geq \min(m, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> ; <i>lwork</i> ≥ 1 . If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the <i>work</i> (1), and no error message related to <i>lwork</i> is issued by xerbla . See <i>Application Notes</i> for the suggested value of <i>lwork</i> .
<i>rwork</i>	REAL for cgesdd DOUBLE PRECISION for zgesdd Workspace array, size at least $\max(1, 5 * \min(m, n))$ if <i>jobz</i> = 'N'. Otherwise, the dimension of <i>rwork</i> must be at least $\max(1, \min(m, n) * \max(5 * \min(m, n) + 7, 2 * \max(m, n) + 2 * \min(m, n) + 1))$. This array is used in complex flavors only.
<i>iwork</i>	INTEGER. Workspace array, size at least $\max(1, 8 * \min(m, n))$.

Output Parameters

<i>a</i>	On exit: If <i>jobz</i> = 'O', then if $m \geq n$, <i>a</i> is overwritten with the first <i>n</i> columns of <i>U</i> (the left singular vectors, stored columnwise). If $m < n$, <i>a</i> is overwritten with the first <i>m</i> rows of <i>V</i> ^T (the right singular vectors, stored rowwise); If <i>jobz</i> ≠ 'O', the contents of <i>a</i> are destroyed.
<i>s</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, size at least $\max(1, \min(m, n))$. Contains the singular values of <i>A</i> sorted so that $s(i) \geq s(i+1)$.
<i>u</i> , <i>vt</i>	REAL for sgesdd DOUBLE PRECISION for dgesdd COMPLEX for cgesdd

DOUBLE COMPLEX for zgesdd.

Arrays:

$u(lu,*)$; the second dimension of u must be at least $\max(1, m)$ if $jobz = 'A'$ or $jobz = 'O'$ and $m < n$.

If $jobz = 'S'$, the second dimension of u must be at least $\max(1, \min(m, n))$.

If $jobz = 'A'$ or $jobz = 'O'$ and $m < n$, u contains the m -by- m orthogonal/unitary matrix U .

If $jobz = 'S'$, u contains the first $\min(m, n)$ columns of U (the left singular vectors, stored columnwise).

If $jobz = 'O'$ and $m \geq n$, or $jobz = 'N'$, u is not referenced.

$vt(lv,*)$; the second dimension of vt must be at least $\max(1, n)$.

If $jobz = 'A'$ or $jobz = 'O'$ and $m \geq n$, vt contains the n -by- n orthogonal/unitary matrix V^T .

If $jobz = 'S'$, vt contains the first $\min(m, n)$ rows of V^T (the right singular vectors, stored rowwise).

If $jobz = 'O'$ and $m < n$, or $jobz = 'N'$, vt is not referenced.

$work(1)$

On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, then ?bdsdc did not converge, updating process failed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine gesdd interface are the following:

a Holds the matrix A of size (m, n) .

s Holds the vector of length $\min(m, n)$.

u Holds the matrix U of size

- (m, m) if $jobz = 'A'$ or $jobz = 'O'$ and $m < n$
- $(m, \min(m, n))$ if $jobz = 'S'$

u is not referenced if $jobz$ is not supplied or if $jobz = 'N'$ or $jobz = 'O'$ and $m \geq n$.

vt Holds the matrix VT of size

- (n, n) if $jobz = 'A'$ or $jobz = 'O'$ and $m \geq n$
- $(\min(m, n), n)$ if $jobz = 'S'$

vt is not referenced if *jobz* is not supplied or if *jobz*='N' or *jobz*='O' and *m* < *n*.

job Must be 'N', 'A', 'S', or 'O'. The default value is 'N'.

Application Notes

For real flavors:

If *jobz* = 'N', *lwork* $\geq 3*\min(m, n) + \max(\max(m, n), 6*\min(m, n))$;

If *jobz* = 'O', *lwork* $\geq 3*(\min(m, n))^2 + \max(\max(m, n), 5*(\min(m, n))^2 + 4*\min(m, n))$;

If *jobz* = 'S' or 'A', *lwork* $\geq \min(m, n)*(6 + 4*\min(m, n)) + \max(m, n)$;

For complex flavors:

If *jobz* = 'N', *lwork* $\geq 2*\min(m, n) + \max(m, n)$;

If *jobz* = 'O', *lwork* $\geq 2*(\min(m, n))^2 + \max(m, n) + 2*\min(m, n)$;

If *jobz* = 'S' or 'A', *lwork* $\geq (\min(m, n))^2 + \max(m, n) + 2*\min(m, n)$;

For good performance, *lwork* should generally be larger.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?gejsv

Computes the singular value decomposition of a real matrix using a preconditioned Jacobi SVD method.

Syntax

FORTRAN 77:

```
call sgejsv(job, jobu, jobv, jobr, jobt, jobp, m, n, a, lda, sva, u, ldu, v, ldv,
work, lwork, iwork, info)
```

```
call dgejsv(job, jobu, jobv, jobr, jobt, jobp, m, n, a, lda, sva, u, ldu, v, ldv,
work, lwork, iwork, info)
```

C:

```
lapack_int LAPACKE_<?>gejsv( int matrix_layout, char joba, char jobu, char jobv, char
jobr, char jobt, char jobp, lapack_int m, lapack_int n, const <datatype>* a,
lapack_int lda, <datatype>* sva, <datatype>* u, lapack_int ldu, <datatype>* v,
lapack_int ldv, <datatype>* stat, lapack_int* istat );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the singular value decomposition (SVD) of a real m -by- n matrix A , where $m \geq n$.

The SVD is written as

$$A = U \Sigma V^T,$$

where Σ is an m -by- n matrix which is zero except for its n diagonal elements, U is an m -by- n (or m -by- m) orthonormal matrix, and V is an n -by- n orthogonal matrix. The diagonal elements of Σ are the singular values of A ; the columns of U and V are the left and right singular vectors of A , respectively. The matrices U and V are computed and stored in the arrays u and v , respectively. The diagonal of Σ is computed and stored in the array sva .

The routine implements a preconditioned Jacobi SVD algorithm. It uses `?geqp3`, `?geqr`, and `?gelqf` as preprocessors and preconditioners. Optionally, an additional row pivoting can be used as a preprocessor, which in some cases results in much higher accuracy. An example is matrix A with the structure $A = D_1 * C * D_2$, where D_1, D_2 are arbitrarily ill-conditioned diagonal matrices and C is a well-conditioned matrix. In that case, complete pivoting in the first QR factorizations provides accuracy dependent on the condition number of C , and independent of D_1, D_2 . Such higher accuracy is not completely understood theoretically, but it works well in practice.

If A can be written as $A = B*D$, with well-conditioned B and some diagonal D , then the high accuracy is guaranteed, both theoretically and in software independent of D . For more details see [[Drmac08-1](#)], [[Drmac08-2](#)].

The computational range for the singular values can be the full range (`UNDERFLOW`,`OVERFLOW`), provided that the machine arithmetic and the BLAS and LAPACK routines called by `?gejsv` are implemented to work in that range. If that is not the case, the restriction for safe computation with the singular values in the range of normalized IEEE numbers is that the spectral condition number $\kappa(A) = \sigma_{\max}(A) / \sigma_{\min}(A)$ does not overflow. This code (`?gejsv`) is best used in this restricted range, meaning that singular values of magnitude below $\|A\|_2 / \text{slamch('O')}$ (for single precision) or $\|A\|_2 / \text{dlamch('O')}$ (for double precision) are returned as zeros. See `jobr` for details on this.

This implementation is slower than the one described in [[Drmac08-1](#)], [[Drmac08-2](#)] due to replacement of some non-LAPACK components, and because the choice of some tuning parameters in the iterative part (`?gesvj`) is left to the implementer on a particular machine.

The rank revealing QR factorization (in this code: `?geqp3`) should be implemented as in [[Drmac08-3](#)].

If m is much larger than n , it is obvious that the initial QRF with column pivoting can be preprocessed by the QRF without pivoting. That well known trick is not used in `?gejsv` because in some cases heavy row weighting can be treated with complete pivoting. The overhead in cases m much larger than n is then only due to pivoting, but the benefits in accuracy have prevailed. You can incorporate this extra QRF step easily and also improve data movement (matrix transpose, matrix copy, matrix transposed copy) - this implementation of `?gejsv` uses only the simplest, naive data movement.

Input Parameters

The data types are given for the Fortran interface, except for `istat`. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

`joba`

CHARACTER*1. Must be '`C`', '`E`', '`F`', '`G`', '`A`', or '`R`'.

Specifies the level of accuracy:

If `joba = 'C'`, high relative accuracy is achieved if $A = B*D$ with well-conditioned B and arbitrary diagonal matrix D . The accuracy cannot be spoiled by column scaling. The accuracy of the computed output depends on the condition of B , and the procedure aims at the best theoretical

accuracy. The relative error $\max_{i=1:N} |d \sigma_i| / \sigma_i$ is bounded by $f(M, N) * \epsilon \operatorname{cond}(B)$, independent of D . The input matrix is preprocessed with the QRF with column pivoting. This initial preprocessing and preconditioning by a rank revealing QR factorization is common for all values of joba . Additional actions are specified as follows:

If $\text{joba} = 'E'$, computation as with ' C ' with an additional estimate of the condition number of B . It provides a realistic error bound.

If $\text{joba} = 'F'$, accuracy higher than in the ' C ' option is achieved, if $A = D1 * C * D2$ with ill-conditioned diagonal scalings $D1, D2$, and a well-conditioned matrix C . This option is advisable, if the structure of the input matrix is not known and relative accuracy is desirable. The input matrix A is preprocessed with QR factorization with full (row and column) pivoting.

If $\text{joba} = 'G'$, computation as with ' F ' with an additional estimate of the condition number of B , where $A = B * D$. If A has heavily weighted rows, using this condition number gives too pessimistic error bound.

If $\text{joba} = 'A'$, small singular values are the noise and the matrix is treated as numerically rank deficient. The error in the computed singular values is bounded by $f(m, n) * \epsilon * \|A\|$. The computed SVD $A = U * S * V^T$ restores A up to $f(m, n) * \epsilon * \|A\|$. This enables the procedure to set all singular values below $n * \epsilon * \|A\|$ to zero.

If $\text{joba} = 'R'$, the procedure is similar to the ' A ' option. Rank revealing property of the initial QR factorization is used to reveal (using triangular factor) a gap $\sigma_{r+1} < \epsilon * \sigma_r$, in which case the numerical rank is declared to be r . The SVD is computed with absolute error bounds, but more accurately than with ' A '.

jobu

CHARACTER*1. Must be ' U ', ' F ', ' W ', or ' N '.

Specifies whether to compute the columns of the matrix U :

If $\text{jobu} = 'U'$, n columns of U are returned in the array u .

If $\text{jobu} = 'F'$, a full set of m left singular vectors is returned in the array u .

If $\text{jobu} = 'W'$, u may be used as workspace of length $m * n$. See the description of u .

If $\text{jobu} = 'N'$, u is not computed.

jobv

CHARACTER*1. Must be ' V ', ' J ', ' W ', or ' N '.

Specifies whether to compute the matrix V :

If $\text{jobv} = 'V'$, n columns of V are returned in the array v ; Jacobi rotations are not explicitly accumulated.

If $\text{jobv} = 'J'$, n columns of V are returned in the array v but they are computed as the product of Jacobi rotations. This option is allowed only if $\text{jobu} \neq n$

If $\text{jobv} = 'W'$, v may be used as workspace of length $n * n$. See the description of v .

If $\text{jobv} = 'N'$, v is not computed.

jobr

CHARACTER*1. Must be ' N ' or ' R '.

Specifies the range for the singular values. If small positive singular values are outside the specified range, they may be set to zero. If A is scaled so that the largest singular value of the scaled matrix is around $\text{sqrt}(\text{big})$, $\text{big} = \text{?lamch('O')}$, the function can remove columns of A whose norm in the scaled matrix is less than $\text{sqrt}(\text{?lamch('S')})$ (for $\text{jobr} = 'R'$), or less than $\text{small} = \text{?lamch('S')}/\text{?lamch('E')}$.

If $\text{jobr} = 'N'$, the function does not remove small columns of the scaled matrix. This option assumes that BLAS and QR factorizations and triangular solvers are implemented to work in that range. If the condition of A is greater than big , use `?gesvj`.

If $\text{jobr} = 'R'$, restricted range for singular values of the scaled matrix A is $[\text{sqrt}(\text{?lamch('S')}, \text{sqrt}(\text{big}))]$, roughly as described above. This option is recommended.

For computing the singular values in the full range $[\text{?lamch('S')}, \text{big}]$, use `?gesvj`.

jobt

CHARACTER*1. Must be 'T' or 'N'.

If the matrix is square, the procedure may determine to use a transposed A if A^{**t} seems to be better with respect to convergence. If the matrix is not square, *jobt* is ignored. This is subject to changes in the future.

The decision is based on two values of entropy over the adjoint orbit of $A^{**t} * A$. See the descriptions of `work(6)` and `work(7)`.

If *jobt* = 'T', the function performs transposition if the entropy test indicates possibly faster convergence of the Jacobi process, if A is taken as input. If A is replaced with A^{**t} , the row pivoting is included automatically.

If *jobt* = 'N', the function attempts no speculations. This option can be used to compute only the singular values, or the full SVD (u , σ , and v). For only one set of singular vectors (u or v), the caller should provide both u and v , as one of the matrices is used as workspace if the matrix A is transposed. The implementer can easily remove this constraint and make the code more complicated. See the descriptions of u and v .

jobp

CHARACTER*1. Must be 'P' or 'N'.

Enables structured perturbations of denormalized numbers. This option should be active if the denormals are poorly implemented, causing slow computation, especially in cases of fast convergence. For details, see [Drmac08-1], [Drmac08-2]. For simplicity, such perturbations are included only when the full SVD or only the singular values are requested. You can add the perturbation for the cases of computing one set of singular vectors.

If *jobp* = 'P', the function introduces perturbation.

If *jobp* = 'N', the function introduces no perturbation.

m

INTEGER. The number of rows of the input matrix A ; $m \geq 0$.

n

INTEGER. The number of columns in the input matrix A ; $n \geq 0$.

a, work, sva, u, v

REAL for `sgejsv`

DOUBLE PRECISION for `dgejsv`.

Array $a(1:\text{da},*)$ is an array containing the m -by- n matrix A .

The second dimension of a must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

sva is a workspace array, its dimension is n .

u is a workspace array, its dimension is $(lDU, *)$; the second dimension of u must be at least $\max(1, n)$.

v is a workspace array, its dimension is $(lDV, *)$; the second dimension of v must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of the array a . Must be at least $\max(1, m)$.

ldu INTEGER. The leading dimension of the array u ; $lDU \geq 1$.

$jobu = 'U'$ or $'F'$ or $'W'$, $lDU \geq m$.

ldv INTEGER. The leading dimension of the array v ; $lDV \geq 1$.

$jobv = 'V'$ or $'J'$ or $'W'$, $lDV \geq n$.

$lwork$ INTEGER.

Length of $work$ to confirm proper allocation of work space. $lwork$ depends on the task performed:

If only σ is needed ($jobu = 'N'$, $jobv = 'N'$) and

- ... no scaled condition estimate is required, then $lwork \geq \max(2*m+n, 4*n+1, 7)$. This is the minimal requirement. For optimal performance (blocked code) the optimal value is $lwork \geq \max(2*m+n, 3*n+(n+1)*nb, 7)$. Here nb is the optimal block size for ?geqp3/?geqrif.

In general, the optimal length $lwork$ is computed as

$lwork \geq \max(2*m+n, n+lwork(sgeqp3), n+lwork(sgeqrif), 7)$ for sgejsv

$lwork \geq \max(2*m+n, n+lwork(dgeqp3), n+lwork(dgeqrif), 7)$ for dgejsv

- ... an estimate of the scaled condition number of A is required ($joba = 'E'$, $'G'$). In this case, $lwork$ is the maximum of the above and $n*n + 4*n$, that is, $lwork \geq \max(2*m+n, n*n+4*n, 7)$. For optimal performance (blocked code) the optimal value is $lwork \geq \max(2*m+n, 3*n+(n+1)*nb, n*n+4*n, 7)$.

In general, the optimal length $lwork$ is computed as

$lwork \geq \max(2*m+n, n+lwork(sgeqp3), n+lwork(sgeqrif), n+n*n + lwork(spocon), 7)$ for sgejsv

$lwork \geq \max(2*m+n, n+lwork(dgeqp3), n+lwork(dgeqrif), n+n*n + lwork(dpocon), 7)$ for dgejsv

If σ and the right singular vectors are needed ($jobv = 'V'$),

- the minimal requirement is $lwork \geq \max(2*m+n, 4*n+1, 7)$.
- for optimal performance, $lwork \geq \max(2*m+n, 3*n+(n+1)*nb, 7)$, where nb is the optimal block size for ?geqp3, ?geqrif, ?gelqf, ?ormlq.

In general, the optimal length $lwork$ is computed as

$lwork \geq \max(2*m+n, n+lwork(sgeqp3), n+lwork(spocon), n+lwork(sgelqf), 2*n+lwork(sgeqr), n+lwork(sormlq)$ for sgejsv

$lwork \geq \max(2*m+n, n+lwork(dgeqp3), n+lwork(dpocon), n+lwork(dgelqf), 2*n+lwork(dgeqr), n+lwork(dormlq)$ for dgejsv

If σ and the left singular vectors are needed

- the minimal requirement is $lwork \geq \max(2*n+m, 4*n+1, 7)$.
- for optimal performance,


```
if jobu = 'U' :: lwork >= max(2*m+n, 3*n+(n+1)*nb, 7),
if jobu = 'F' :: lwork >= max(2*m+n, 3*n+(n+1)*nb, n+m*nb, 7),
where nb is the optimal block size for ?geqp3, ?geqr, ?ormlq . In
general, the optimal length lwork is computed as
```

$lwork \geq \max(2*m+n, n+lwork(sgeqp3), n+lwork(spocon), 2*n+lwork(sgeqr), n+lwork(sormlq)$ for sgejsv

$lwork \geq \max(2*m+n, n+lwork(dgeqp3), n+lwork(dpocon), 2*n+lwork(dgeqr), n+lwork(dormlq)$ for dgejsv

Here $lwork(?ormlq)$ equals $n*nb$ (for $jobu = 'U'$) or $m*nb$ (for $jobu = 'F'$)

If full SVD is needed ($jobu = 'U'$ or $'F'$) and

- if $jobv = 'V'$,
the minimal requirement is $lwork \geq \max(2*m+n, 6*n+2*n*n)$
- if $jobv = 'J'$,
the minimal requirement is $lwork \geq \max(2*m+n, 4*n+n*n, 2*n+n*n+6)$
- For optimal performance, $lwork$ should be additionally larger than $n+m*nb$, where nb is the optimal block size for $?ormlq$.

$iwork$

INTEGER. Workspace array, DIMENSION $\max(3, m+3*n)$.

Output Parameters

sva

On exit:

For $work(1)/work(2) = \text{one}$: the singular values of A . During the computation sva contains Euclidean column norms of the iterated matrices in the array a .

For $work(1) \neq work(2)$: the singular values of A are $(work(1)/work(2)) * sva(1:n)$. This factored form is used if $\sigma_{\max}(A)$ overflows or if small singular values have been saved from underflow by scaling the input matrix A .

$jobr = 'R'$, some of the singular values may be returned as exact zeros obtained by 'setting to zero' because they are below the numerical rank threshold or are denormalized numbers.

u

On exit:

If $\text{jobu} = \text{'U'}$, contains the m -by- n matrix of the left singular vectors.

If $\text{jobu} = \text{'F'}$, contains the m -by- m matrix of the left singular vectors, including an orthonormal basis of the orthogonal complement of the range of A .

If $\text{jobu} = \text{'W'}$ and $\text{jobv} = \text{'V'}$, $\text{jobt} = \text{'T'}$, and $m = n$, then u is used as workspace if the procedure replaces A with A^{**t} . In that case, v is computed in u as left singular vectors of A^{**t} and copied back to the v array. This ' W ' option is just a reminder to the caller that in this case u is reserved as workspace of length $n*n$.

If $\text{jobu} = \text{'N'}$, u is not referenced.

v

On exit:

If $\text{jobv} = \text{'V'}$ or 'J' , contains the n -by- n matrix of the right singular vectors.

If $\text{jobv} = \text{'W'}$ and $\text{jobv} = \text{'U'}$, $\text{jobt} = \text{'T'}$, and $m = n$, then v is used as workspace if the procedure replaces A with A^{**t} . In that case, u is computed in v as right singular vectors of A^{**t} and copied back to the u array. This ' W ' option is just a reminder to the caller that in this case v is reserved as workspace of length $n*n$.

If $\text{jobv} = \text{'N'}$, v is not referenced.

$work$

On exit,

$work(1) = scale = work(2)/work(1)$ is the scaling factor such that $scale*sva(1:n)$ are the computed singular values of A . See the description of $sva()$.

$work(2) =$ see the description of $work(1)$.

$work(3) = sconda$ is an estimate for the condition number of column equilibrated A . If $\text{joba} = \text{'E'}$ or 'G' , $sconda$ is an estimate of $\sqrt{\|(R^{**t} * R)^{**(-1)}\|_1}$. It is computed using $?pocon$. It holds $n^{**(-1/4)} * sconda \leq \|R^{**(-1)}\|_2 \leq n^{*(1/4)} * sconda$, where R is the triangular factor from the QRF of A . However, if R is truncated and the numerical rank is determined to be strictly smaller than n , $sconda$ is returned as -1, indicating that the smallest singular values might be lost.

If full SVD is needed, the following two condition numbers are useful for the analysis of the algorithm. They are provided for a user who is familiar with the details of the method.

$work(4) =$ an estimate of the scaled condition number of the triangular factor in the first QR factorization.

$work(5) =$ an estimate of the scaled condition number of the triangular factor in the second QR factorization.

The following two parameters are computed if $\text{jobt} = \text{'T'}$. They are provided for a user who is familiar with the details of the method.

$work(6) =$ the entropy of $A^{**t}*A ::$ this is the Shannon entropy of $\text{diag}(A^{**t}*A) / \text{Trace}(A^{**t}*A)$ taken as point in the probability simplex.

$work(7) =$ the entropy of $A*A^{**t}$.

iwork (Fortran), *istat* (C) INTEGER. On exit,

iwork(1)/*istat*[0] = the numerical rank determined after the initial QR factorization with pivoting. See the descriptions of *joba* and *jobr*.

iwork(2)/*istat*[1] = the number of the computed nonzero singular value.

iwork(3)/*istat*[2] = if nonzero, a warning message. If *iwork*(3)/*istat*[2]=1, some of the column norms of *A* were denormalized floats. The requested high accuracy is not warranted by the data.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, the function did not converge in the maximal number of sweeps. The computed values may be inaccurate.

See Also

[?geqp3](#)
[?geqrf](#)
[?gelqf](#)
[?gesvj](#)
[?lamch](#)
[?pocon](#)
[?ormlq](#)

[?gesvj](#)

Computes the singular value decomposition of a real matrix using Jacobi plane rotations.

Syntax

FORTRAN 77:

```
call sgesvj(job, jobu, jobv, m, n, a, lda, sva, mv, v, ldv, work, lwork, info)
call dgesvj(job, jobu, jobv, m, n, a, lda, sva, mv, v, ldv, work, lwork, info)
```

C:

```
lapack_int LAPACKE_<?>gesvj( int matrix_layout, char joba, char jobu, char jobv,
lapack_int m, lapack_int n, <datatype>* a, lapack_int lda, <datatype>* sva, lapack_int
mv, <datatype>* v, lapack_int ldv, <datatype>* stat );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the singular value decomposition (SVD) of a real m -by- n matrix *A*, where $m \geq n$.

The SVD of *A* is written as

$$A = U^* \Sigma^* V^*,$$

where Σ is an m -by- n diagonal matrix, U is an m -by- n orthonormal matrix, and V is an n -by- n orthogonal matrix. The diagonal elements of Σ are the singular values of A ; the columns of U and V are the left and right singular vectors of A , respectively. The matrices U and V are computed and stored in the arrays u and v , respectively. The diagonal of Σ is computed and stored in the array sva .

The n -by- n orthogonal matrix V is obtained as a product of Jacobi plane rotations. The rotations are implemented as fast scaled rotations of Anda and Park [AndaPark94]. In the case of underflow of the Jacobi angle, a modified Jacobi transformation of Drmac ([Drmac08-4]) is used. Pivot strategy uses column interchanges of de Rijk ([deRijk98]). The relative accuracy of the computed singular values and the accuracy of the computed singular vectors (in angle metric) is as guaranteed by the theory of Demmel and Veselic [Demmel92]. The condition number that determines the accuracy in the full rank case is essentially

$$(\min_i d_i) \cdot \kappa(A \cdot D)$$

where $\kappa(\cdot)$ is the spectral condition number. The best performance of this Jacobi SVD procedure is achieved if used in an accelerated version of Drmac and Veselic [Drmac08-1], [Drmac08-2]. Some tunning parameters (marked with TP) are available for the implementer.

The computational range for the nonzero singular values is the machine number interval (`UNDERFLOW`, `OVERFLOW`). In extreme cases, even denormalized singular values can be computed with the corresponding gradual loss of accurate digit.

Input Parameters

The data types are given for the Fortran interface, except for `stat`. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>joba</code>	CHARACTER*1. Must be ' <code>L</code> ', ' <code>U</code> ' or ' <code>G</code> '. Specifies the structure of A : If <code>joba</code> = ' <code>L</code> ', the input matrix A is lower triangular. If <code>joba</code> = ' <code>U</code> ', the input matrix A is upper triangular. If <code>joba</code> = ' <code>G</code> ', the input matrix A is a general m -by- n , $m \geq n$.
<code>jobu</code>	CHARACTER*1. Must be ' <code>U</code> ', ' <code>C</code> ' or ' <code>N</code> '. Specifies whether to compute the left singular vectors (columns of U): If <code>jobu</code> = ' <code>U</code> ', the left singular vectors corresponding to the nonzero singular values are computed and returned in the leading columns of A . See more details in the description of <code>a</code> . The default numerical orthogonality threshold is set to approximately <code>TOL=CTOL*EPS</code> , <code>CTOL=sqrt(m)</code> , <code>EPS = ?lamch('E')</code> If <code>jobu</code> = ' <code>C</code> ', analogous to <code>jobu</code> = ' <code>U</code> ', except that you can control the level of numerical orthogonality of the computed left singular vectors. <code>TOL</code> can be set to <code>TOL=CTOL*EPS</code> , where <code>CTOL</code> is given on input in the array <code>work</code> . No <code>CTOL</code> smaller than <code>ONE</code> is allowed. <code>CTOL</code> greater than <code>1 / EPS</code> is meaningless. The option ' <code>C</code> ' can be used if $m * EPS$ is satisfactory orthogonality of the computed left singular vectors, so <code>CTOL=m</code> could save a few sweeps of Jacobi rotations. See the descriptions of <code>a</code> and <code>work(1)</code> . If <code>jobu</code> = ' <code>N</code> ', u is not computed. However, see the description of <code>a</code> .
<code>jobv</code>	CHARACTER*1. Must be ' <code>V</code> ', ' <code>A</code> ' or ' <code>N</code> '.

Specifies whether to compute the right singular vectors, that is, the matrix V :

If $jobv = 'V'$, the matrix V is computed and returned in the array v .

If $jobv = 'A'$, the Jacobi rotations are applied to the mv -by- n array v . In other words, the right singular vector matrix V is not computed explicitly, instead it is applied to an mv -by- n matrix initially stored in the first mv rows of V .

If $jobv = 'N'$, the matrix V is not computed and the array v is not referenced.

m

INTEGER. The number of rows of the input matrix A .

$1/\text{slamch('E')} > m \geq 0$ for `sgesvj`.

$1/\text{dlamch('E')} > m \geq 0$ for `dgesvj`.

n

INTEGER. The number of columns in the input matrix A ; $n \geq 0$.

$a, work, sva, v$

REAL for `sgesvj`

DOUBLE PRECISION for `dgesvj`.

Array $a(1:da,*)$ is an array containing the m -by- n matrix A .

The second dimension of a is $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(4, m+n)$.

If $jobu = 'C'$, $work(1)=\text{CTOL}$, where CTOL defines the threshold for convergence. The process stops if all columns of A are mutually orthogonal up to $\text{CTOL}*\text{EPS}$, $\text{EPS}=\text{lamch('E')}$. It is required that $\text{CTOL} \geq 1$, that is, it is not allowed to force the routine to obtain orthogonality below ϵ .

sva is a workspace array, its dimension is n .

v is a workspace array, its dimension is $(1:dv,*)$; the second dimension of v must be at least $\max(1, n)$.

lda

INTEGER. The leading dimension of the array a . Must be at least $\max(1, m)$.

mv

INTEGER.

$jobv = 'A'$, the product of Jacobi rotations in `gesvj` is applied to the first mv rows of v . See the description of $jobv$.

ldv

INTEGER. The leading dimension of the array v ; $ldv \geq 1$.

$jobv = 'V'$, $ldv \geq \max(1, n)$.

$jobv = 'A'$, $ldv \geq \max(1, mv)$.

$lwork$

INTEGER.

Length of $work$, $work \geq \max(6, m+n)$.

Output Parameters

a

On exit:

If $jobu = 'U'$ or $jobu = 'C'$:

- if $info = 0$, the leading columns of A contain left singular vectors corresponding to the computed singular values of a that are above the underflow threshold $?lamch('S')$, that is, non-zero singular values. The number of the computed non-zero singular values is returned in $work(2)$. Also see the descriptions of sva and $work$. The computed columns of u are mutually numerically orthogonal up to approximately $TOL=\sqrt{m} * EPS$ (default); or $TOL=CTOL*EPS$ $jobu = 'C'$, see the description of $jobu$.
- if $info > 0$, the procedure $?gesvj$ did not converge in the given number of iterations (sweeps). In that case, the computed columns of u may not be orthogonal up to TOL . The output u (stored in a), $sigma$ (given by the computed singular values in $sva(1:n)$) and v is still a decomposition of the input matrix A in the sense that the residual $\|a - scale*u*sigma*v**t\|_2 / \|a\|_2$ is small.

If $jobu = 'N'$:

- if $info = 0$, note that the left singular vectors are 'for free' in the one-sided Jacobi SVD algorithm. However, if only the singular values are needed, the level of numerical orthogonality of u is not an issue and iterations are stopped when the columns of the iterated matrix are numerically orthogonal up to approximately $m*EPS$. Thus, on exit, a contains the columns of u scaled with the corresponding singular values.
- if $info > 0$, the procedure $?gesvj$ did not converge in the given number of iterations (sweeps).

sva

On exit:

If $info = 0$, depending on the value $scale = work(1)$, where $scale$ is the scaling factor:

- if $scale = 1$, $sva(1:n)$ contains the computed singular values of a . During the computation, sva contains the Euclidean column norms of the iterated matrices in the array a .
- if $scale \neq 1$, the singular values of a are $scale*sva(1:n)$, and this factored representation is due to the fact that some of the singular values of a might underflow or overflow.

If $info > 0$, the procedure $?gesvj$ did not converge in the given number of iterations (sweeps) and $scale*sva(1:n)$ may not be accurate.

v

On exit:

If $jobv = 'V'$, contains the n -by- n matrix of the right singular vectors.

If $jobv = 'A'$, then v contains the product of the computed right singular vector matrix and the initial matrix in the array v .

If $jobv = 'N'$, v is not referenced.

$work$ (Fortarn), $stat$ (C)

On exit,

$work(1)/stat[0] = scale$ is the scaling factor such that $scale*sva(1:n)$ are the computed singular values of A . See the description of $sva()$.

$work(2)/stat[1]$ is the number of the computed nonzero singular value.

work(3)/*stat*[2] is the number of the computed singular values that are larger than the underflow threshold.

work(4)/*stat*[3] is the number of sweeps of Jacobi rotations needed for numerical convergence.

work(5)/*stat*[4] = max_{i.NE.j} |COS(A(:,i),A(:,j))| in the last sweep. This is useful information in cases when ?gesvj did not converge, as it can be used to estimate whether the output is still useful and for post festum analysis.

work(6)/*stat*[5] is the largest absolute value over all sines of the Jacobi rotation angles in the last sweep. It can be useful in a post festum analysis.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* > 0, the function did not converge in the maximal number (30) of sweeps. The output may still be useful. See the description of *work*.

See Also

?lamch

?ggsvd

Computes the generalized singular value decomposition of a pair of general rectangular matrices.

Syntax

FORTRAN 77:

```
call sggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, iwork, info)

call dggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, iwork, info)

call cggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, rwork, iwork, info)

call zggsvd(jobu, jobv, jobq, m, n, p, k, l, a, lda, b, ldb, alpha, beta, u, ldu, v,
ldv, q, ldq, work, rwork, iwork, info)
```

Fortran 95:

```
call ggsvd(a, b, alpha, beta [, k] [, l] [, u] [, v] [, q] [, iwork] [, info])
```

C:

```
lapack_int LAPACKE_sggsvd( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int* k, lapack_int* l, float* a,
lapack_int lda, float* b, lapack_int ldb, float* alpha, float* beta, float* u,
lapack_int ldu, float* v, lapack_int ldv, float* q, lapack_int ldq, lapack_int*
iwork );
```

```

lapack_int LAPACKE_dggsvd( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int* k, lapack_int* l, double* a,
lapack_int lda, double* b, lapack_int ldb, double* alpha, double* beta, double* u,
lapack_int ldu, double* v, lapack_int ldv, double* q, lapack_int ldq, lapack_int*
iwork );

lapack_int LAPACKE_cggsvd( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int* k, lapack_int* l,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
float* alpha, float* beta, lapack_complex_float* u, lapack_int ldu,
lapack_complex_float* v, lapack_int ldv, lapack_complex_float* q, lapack_int ldq,
lapack_int* iwork );

lapack_int LAPACKE_zggsvd( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int* k, lapack_int* l,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
double* alpha, double* beta, lapack_complex_double* u, lapack_int ldu,
lapack_complex_double* v, lapack_int ldv, lapack_complex_double* q, lapack_int ldq,
lapack_int* iwork );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes the generalized singular value decomposition (GSVD) of an m -by- n real/complex matrix A and p -by- n real/complex matrix B :

$$U'^* A^* Q = D_1 \begin{pmatrix} 0 & R \end{pmatrix}, V'^* B^* Q = D_2 \begin{pmatrix} 0 & R \end{pmatrix},$$

where U , V and Q are orthogonal/unitary matrices and U' , V' mean transpose/conjugate transpose of U and V respectively.

Let $k+l$ = the effective numerical rank of the matrix $(A', B')'$, then R is a $(k+l)$ -by- $(k+l)$ nonsingular upper triangular matrix, D_1 and D_2 are m -by- $(k+l)$ and p -by- $(k+l)$ "diagonal" matrices and of the following structures, respectively:

If $m-k-l \geq 0$,

... 0 0 0 0 0 0 ...

$$D_2 = \begin{matrix} & k & l \\ p - l & \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{matrix}$$

$$\begin{array}{ccccccccc} & \cdots & & \cdots & & & & & \\ & \vdots & \\ & \cdots & & \cdots & & & & & \end{array}$$

where

$C = \text{diag}(\alpha(K+1), \dots, \alpha(K+1))$

$S = \text{diag}(\beta(K+1), \dots, \beta(K+1))$

$C^2 + S^2 = I$

R is stored in $a(1:k+l, n-k-l+1:n)$ on exit.

If $m-k-l < 0$,

$$\begin{array}{ccccccccc} & \cdots & & \cdots & & & & & \\ & \vdots & \\ & \cdots & & \cdots & & & & & \end{array}$$

$$\begin{array}{ccccccccc} & \cdots & & \cdots & & & & & \\ & \vdots & \\ & \cdots & & \cdots & & & & & \end{array}$$

$$(0 \quad R) = \begin{pmatrix} n - k - l & k & m - k & k + l - m \\ & k & R_{11} & R_{12} & R_{13} \\ & m - k & 0 & R_{22} & R_{23} \\ & k + l - m & 0 & 0 & R_{33} \end{pmatrix}$$

where

$C = \text{diag}(\alpha(K+1), \dots, \alpha(m))$,

$S = \text{diag}(\beta(K+1), \dots, \beta(m))$,

$C_2 + S_2 = I$

$$\left[\begin{array}{cccccc} & \cdots & & & & & \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \\ & \cdots & & \cdots & & & \end{array} \right]$$

On exit, $\begin{bmatrix} & \cdots & & & & & \\ & \vdots & \vdots & \vdots & \vdots & \vdots & \\ & \cdots & & \cdots & & & \end{bmatrix}$ is stored in $a(1:m, n-k-l+1:n)$ and R_{33} is stored in $b(m-k+1:l, n+m-k-l+1:n)$.

The routine computes C , S , R , and optionally the orthogonal/unitary transformation matrices U , V and Q .

In particular, if B is an n -by- n nonsingular matrix, then the GSVD of A and B implicitly gives the SVD of A^*B^{-1} :

$$A^*B^{-1} = U^* (D_1^* D_2^{-1}) * V^*.$$

If (A', B') ' has orthonormal columns, then the GSVD of A and B is also equal to the CS decomposition of A and B . Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

$$A'^* A^* x = \lambda * B'^* B^* x.$$

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobu</i>	CHARACTER*1. Must be 'U' or 'N'. If <i>jobu</i> = 'U', orthogonal/unitary matrix U is computed. If <i>jobu</i> = 'N', U is not computed.
<i>jobv</i>	CHARACTER*1. Must be 'V' or 'N'. If <i>jobv</i> = 'V', orthogonal/unitary matrix V is computed. If <i>jobv</i> = 'N', V is not computed.
<i>jobq</i>	CHARACTER*1. Must be 'Q' or 'N'. If <i>jobq</i> = 'Q', orthogonal/unitary matrix Q is computed. If <i>jobq</i> = 'N', Q is not computed.
<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrices A and B ($n \geq 0$).
<i>p</i>	INTEGER. The number of rows of the matrix B ($p \geq 0$).
<i>a</i> , <i>b</i> , <i>work</i>	REAL for sggsvd DOUBLE PRECISION for dggsvd COMPLEX for cggsvd DOUBLE COMPLEX for zggsvd. Arrays: <i>a</i> (<i>lda</i> ,*) contains the m -by- n matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb</i> ,*) contains the p -by- n matrix B . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array. The dimension of <i>work</i> must be at least $\max(3n, m, p) + n$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; at least $\max(1, p)$.

<i>ldu</i>	INTEGER. The leading dimension of the array <i>u</i> . <i>ldu</i> $\geq \max(1, m)$ if <i>jobu</i> = 'U'; <i>ldu</i> ≥ 1 otherwise.
<i>ldv</i>	INTEGER. The leading dimension of the array <i>v</i> . <i>ldv</i> $\geq \max(1, p)$ if <i>jobv</i> = 'V'; <i>ldv</i> ≥ 1 otherwise.
<i>ldq</i>	INTEGER. The leading dimension of the array <i>q</i> . <i>ldq</i> $\geq \max(1, n)$ if <i>jobq</i> = 'Q'; <i>ldq</i> ≥ 1 otherwise.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.
<i>rwork</i>	REAL for <i>cggsvd</i> DOUBLE PRECISION for <i>zggsvd</i> . Workspace array, DIMENSION at least $\max(1, 2n)$. Used in complex flavors only.

Output Parameters

<i>k, l</i>	INTEGER. On exit, <i>k</i> and <i>l</i> specify the dimension of the subblocks. The sum <i>k+l</i> is equal to the effective numerical rank of (A', B') .
<i>a</i>	On exit, <i>a</i> contains the triangular matrix <i>R</i> or part of <i>R</i> .
<i>b</i>	On exit, <i>b</i> contains part of the triangular matrix <i>R</i> if $m-k-l < 0$.
<i>alpha, beta</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays, DIMENSION at least $\max(1, n)$ each. Contain the generalized singular value pairs of <i>A</i> and <i>B</i> : <i>alpha</i> (1:k) = 1, <i>beta</i> (1:k) = 0, and if $m-k-l \geq 0$, <i>alpha</i> (k+1:k+1) = <i>C</i> , <i>beta</i> (k+1:k+1) = <i>S</i> , or if $m-k-l < 0$, <i>alpha</i> (k+1:m) = <i>C</i> , <i>alpha</i> (m+1:k+1) = 0 <i>beta</i> (k+1:m) = <i>S</i> , <i>beta</i> (m+1:k+1) = 1 and <i>alpha</i> (k+l+1:n) = 0 <i>beta</i> (k+l+1:n) = 0.
<i>u, v, q</i>	REAL for <i>sggsvd</i> DOUBLE PRECISION for <i>dggsvd</i> COMPLEX for <i>cggsvd</i>

DOUBLE COMPLEX for zggsvd.

Arrays:

$u(lu,*)$; the second dimension of u must be at least $\max(1, m)$.

If $jobu = 'U'$, u contains the m -by- m orthogonal/unitary matrix U .

If $jobu = 'N'$, u is not referenced.

$v(lv,*)$; the second dimension of v must be at least $\max(1, p)$.

If $jobv = 'V'$, v contains the p -by- p orthogonal/unitary matrix V .

If $jobv = 'N'$, v is not referenced.

$q(lq,*)$; the second dimension of q must be at least $\max(1, n)$.

If $jobq = 'Q'$, q contains the n -by- n orthogonal/unitary matrix Q .

If $jobq = 'N'$, q is not referenced.

$iwork$ On exit, $iwork$ stores the sorting information.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = 1$, the Jacobi-type procedure failed to converge. For further details, see subroutine [tgsja](#).

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggsvd` interface are the following:

a Holds the matrix A of size (m, n) .

b Holds the matrix B of size (p, n) .

$alpha$ Holds the vector of length n .

$beta$ Holds the vector of length n .

u Holds the matrix U of size (m, m) .

v Holds the matrix V of size (p, p) .

q Holds the matrix Q of size (n, n) .

$iwork$ Holds the vector of length n .

$jobu$ Restored based on the presence of the argument u as follows:

$jobu = 'U'$, if u is present, $jobu = 'N'$, if u is omitted.

$jobv$ Restored based on the presence of the argument v as follows:

$jobz = 'V'$, if v is present,

	<i>jobz</i> = 'N', if <i>v</i> is omitted.
<i>jobq</i>	Restored based on the presence of the argument <i>q</i> as follows: <i>jobz</i> = 'Q', if <i>q</i> is present, <i>jobz</i> = 'N', if <i>q</i> is omitted.

Cosine-Sine Decomposition

This section describes LAPACK driver routines for computing the *cosine-sine decomposition* (CS decomposition). You can also call the corresponding computational routines to perform the same task.

The computation has the following phases:

1. The matrix is reduced to a bidiagonal block form.
2. The blocks are simultaneously diagonalized using techniques from the bidiagonal SVD algorithms.

[Table "Driver Routines for Cosine-Sine Decomposition \(CSD\)"](#) lists LAPACK routines (FORTRAN 77 interface) that perform CS decomposition of matrices. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Computational Routines for Cosine-Sine Decomposition (CSD)

Operation	Real matrices	Complex matrices
Compute the CS decomposition of a block-partitioned orthogonal matrix	orcisd uncsd orcisd2by1 uncsd2by1	
Compute the CS decomposition of a block-partitioned unitary matrix		orcisd uncsd

See Also

[Cosine-Sine Decomposition](#)

?orcisd/?uncsd

Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

Syntax

FORTRAN 77:

```
call sorcsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t,
work, lwork, iwork, info )

call dorcsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t,
work, lwork, iwork, info )

call cuncsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t,
work, lwork, rwork, lrwork, iwork, info )

call zuncsd( jobu1, jobu2, jobv1t, jobv2t, trans, signs, m, p, q, x11, ldx11, x12,
ldx12, x21, ldx21, x22, ldx22, theta, u1, ldu1, u2, ldu2, v1t, ldv1t, v2t, ldv2t,
work, lwork, rwork, lrwork, iwork, info )
```

Fortran 95:

```
call orcisd( x11,x12,x21,x22,theta,u1,u2,v1t,v2t[,jobu1][,jobu2][,jobv1t][,jobv2t]
[,trans][,signs][,info] )
```

```
call uncisd( x11,x12,x21,x22,theta,u1,u2,v1t,v2t[,jobu1][,jobu2][,jobv1t][,jobv2t]
[,trans][,signs][,info] )
```

C:

```
lapack_int LAPACKE_sorcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q, float* x11,
lapack_int ldx11, float* x12, lapack_int ldx12, float* x21, lapack_int ldx21, float*
x22, lapack_int ldx22, float* theta, float* u1, lapack_int ldu1, float* u2, lapack_int
ldu2, float* v1t, lapack_int ldv1t, float* v2t, lapack_int ldv2t );

lapack_int LAPACKE_dorcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q, double* x11,
lapack_int ldx11, double* x12, lapack_int ldx12, double* x21, lapack_int ldx21,
double* x22, lapack_int ldx22, double* theta, double* u1, lapack_int ldu1, double* u2,
lapack_int ldu2, double* v1t, lapack_int ldv1t, double* v2t, lapack_int ldv2t );

lapack_int LAPACKE_cuncsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q,
lapack_complex_float* x11, lapack_int ldx11, lapack_complex_float* x12, lapack_int
ldx12, lapack_complex_float* x21, lapack_int ldx21, lapack_complex_float* x22,
lapack_int ldx22, float* theta, lapack_complex_float* u1, lapack_int ldu1,
lapack_complex_float* u2, lapack_int ldu2, lapack_complex_float* v1t, lapack_int ldv1t,
lapack_complex_float* v2t, lapack_int ldv2t );

lapack_int LAPACKE_zuncsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char
jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q,
lapack_complex_double* x11, lapack_int ldx11, lapack_complex_double* x12, lapack_int
ldx12, lapack_complex_double* x21, lapack_int ldx21, lapack_complex_double* x22,
lapack_int ldx22, double* theta, lapack_complex_double* u1, lapack_int ldu1,
lapack_complex_double* u2, lapack_int ldu2, lapack_complex_double* v1t, lapack_int ldv1t,
lapack_complex_double* v2t, lapack_int ldv2t );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routines ?orcsd/?uncsd compute the CS decomposition of an m -by- m partitioned orthogonal matrix X :

$$X = \begin{pmatrix} x_{11} & | & x_{12} \\ x_{21} & | & x_{22} \end{pmatrix} = \begin{pmatrix} u_1 & | & u_2 \end{pmatrix} \begin{pmatrix} I & 0 & 0|0 & 0 & 0 \\ 0 & C & 0|0 & -S & 0 \\ 0 & 0 & 0|0 & 0 & -I \\ 0 & 0 & 0|I & 0 & 0 \\ 0 & S & 0|0 & C & 0 \\ 0 & 0 & I|0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_1 & | & v_2 \end{pmatrix}^T$$

or unitary matrix:

$$X = \begin{pmatrix} x_{11} & | & x_{12} \\ x_{21} & | & x_{22} \end{pmatrix} = \begin{pmatrix} u_1 & | & \\ & | & u_2 \end{pmatrix} \begin{pmatrix} I & 0 & 0|0 & 0 & 0 \\ 0 & C & 0|0 & -S & 0 \\ 0 & 0 & 0|0 & 0 & -I \\ 0 & 0 & 0|I & 0 & 0 \\ 0 & S & 0|0 & C & 0 \\ 0 & 0 & I|0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_1 & | & \\ & | & v_2 \end{pmatrix}^H$$

x_{11} is p -by- q . The orthogonal/unitary matrices u_1 , u_2 , v_1 , and v_2 are p -by- p , $(m-p)$ -by- $(m-p)$, q -by- q , $(m-q)$ -by- $(m-q)$, respectively. C and S are r -by- r nonnegative diagonal matrices satisfying $C^2 + S^2 = I$, in which $r = \min(p, m-p, q, m-q)$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobu1</i>	CHARACTER. If equals <i>Y</i> , then u_1 is computed. Otherwise, u_1 is not computed.
<i>jobu2</i>	CHARACTER. If equals <i>Y</i> , then u_2 is computed. Otherwise, u_2 is not computed.
<i>jobv1t</i>	CHARACTER. If equals <i>Y</i> , then v_1^t is computed. Otherwise, v_1^t is not computed.
<i>jobv2t</i>	CHARACTER. If equals <i>Y</i> , then v_2^t is computed. Otherwise, v_2^t is not computed.
<i>trans</i>	CHARACTER = 'T': $x, u_1, u_2, v_1^t, v_2^t$ are stored in row-major order. otherwise $x, u_1, u_2, v_1^t, v_2^t$ are stored in column-major order.
<i>signs</i>	CHARACTER = 'O': The lower-left block is made nonpositive (the "other" convention). otherwise The upper-right block is made nonpositive (the "default" convention).
<i>m</i>	INTEGER. The number of rows and columns of the matrix X .
<i>p</i>	INTEGER. The number of rows in x_{11} and x_{12} . $0 \leq p \leq m$.
<i>q</i>	INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq m$.
<i>x</i>	REAL for <i>sorcisd</i> DOUBLE PRECISION for <i>dorcisd</i>

COMPLEX **for** cuncsd
 DOUBLE COMPLEX **for** zuncsd
Array, DIMENSION (Idx,m).
 On entry, the orthogonal/unitary matrix whose CSD is desired.

lidx INTEGER. The leading dimension of the array X . $lidx \geq \max(1, m)$.

ldu1 INTEGER. The leading dimension of the array u_1 . If $jobu1 = 'Y'$, $ldu1 \geq \max(1, p)$.

ldu2 INTEGER. The leading dimension of the array u_2 . If $jobu2 = 'Y'$, $ldu2 \geq \max(1, m-p)$.

ldv1t INTEGER. The leading dimension of the array $v1t$. If $jobv1t = 'Y'$, $ldv1t \geq \max(1, q)$.

ldv2t INTEGER. The leading dimension of the array $v2t$. If $jobv2t = 'Y'$, $ldv2t \geq \max(1, m-q)$.

work REAL **for** sorcsd
 DOUBLE PRECISION **for** dorcsd
 COMPLEX **for** cuncsd
 DOUBLE COMPLEX **for** zuncsd
Workspace array, DIMENSION ($\max(1, lwork)$).

lwork INTEGER. The size of the *work* array. Constraints:
 If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

rwork REAL **for** cuncsd
 DOUBLE PRECISION **for** zuncsd
Workspace array, DIMENSION ($\max(1, lrwork)$).

lrwork INTEGER. The size of the *rwork* array. Constraints:
 If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *rwork* array, returns this value as the first entry of the *rwork* array, and no error message related to *lrwork* is issued by xerbla.

iwork INTEGER. Workspace array, dimension m .

Output Parameters

theta REAL **for** sorcsd
 DOUBLE PRECISION **for** dorcsd
 COMPLEX **for** cuncsd

```

DOUBLE COMPLEX for zuncsd
Array, DIMENSION (r), in which r = min (p, m-p, q, m-q).
C = diag( cos(theta(1)), ..., cos(theta(r)) ), and
S = diag( sin(theta(1)), ..., sin(theta(r)) ).

u1
REAL for sorcsd
DOUBLE PRECISION for dorcsd
COMPLEX for cuncsd
DOUBLE COMPLEX for zuncsd
Array, DIMENSION (p).
If jobu1 = 'Y', u1 contains the p-by-p orthogonal/unitary matrix u1.

u2
REAL for sorcsd
DOUBLE PRECISION for dorcsd
COMPLEX for cuncsd
DOUBLE COMPLEX for zuncsd
Array, DIMENSION (ldu2,m-p).
If jobu2 = 'Y', u2 contains the (m-p)-by-(m-p) orthogonal/unitary matrix
u2.

v1t
REAL for sorcsd
DOUBLE PRECISION for dorcsd
COMPLEX for cuncsd
DOUBLE COMPLEX for zuncsd
Array, DIMENSION (ldv1t,q).
If jobv1t = 'Y', v1t contains the q-by-q orthogonal matrix v1T or unitary
matrix v1H.

v2t
REAL for sorcsd
DOUBLE PRECISION for dorcsd
COMPLEX for cuncsd
DOUBLE COMPLEX for zuncsd
Array, DIMENSION (ldv2t,m-q).
If jobv2t = 'Y', v2t contains the (m-q)-by-(m-q) orthogonal matrix v2T or
unitary matrix v2H.

work
On exit,
If info = 0,      work(1) returns the optimal lwork.
If info > 0,      work(2:r) contains the values phi(1), ...,
phi(r-1) that, together with theta(1), ...,
theta(r) define the matrix in intermediate

```

bidiagonal-block form remaining after nonconvergence. *info* specifies the number of nonzero *phi*'s.

<i>rwork</i>	On exit,	
	If <i>info</i> = 0,	<i>rwork</i> (1) returns the optimal <i>lrwork</i> .
	If <i>info</i> > 0,	<i>rwork</i> (2: <i>r</i>) contains the values <i>phi</i> (1), ..., <i>phi</i> (<i>r</i> -1) that, together with <i>theta</i> (1), ..., <i>theta</i> (<i>r</i>) define the matrix in intermediate bidiagonal-block form remaining after nonconvergence. <i>info</i> specifies the number of nonzero <i>phi</i> 's.
<i>info</i>	INTEGER.	
	= 0: successful exit	
	< 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument has an illegal value	
	> 0: ?bbcisd did not converge. See the description of <i>work</i> above for details.	

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine ?orcisd/?uncsd interface are as follows:

<i>x11</i>	Holds the block of matrix <i>X</i> of size (<i>p</i> , <i>q</i>).
<i>x12</i>	Holds the block of matrix <i>X</i> of size (<i>p</i> , <i>m</i> - <i>q</i>).
<i>x21</i>	Holds the block of matrix <i>X</i> of size (<i>m</i> - <i>p</i> , <i>q</i>).
<i>x22</i>	Holds the block of matrix <i>X</i> of size (<i>m</i> - <i>p</i> , <i>m</i> - <i>q</i>).
<i>theta</i>	Holds the vector of length <i>r</i> = min(<i>p</i> , <i>m</i> - <i>p</i> , <i>q</i> , <i>m</i> - <i>q</i>).
<i>u1</i>	Holds the matrix of size (<i>p</i> , <i>p</i>).
<i>u2</i>	Holds the matrix of size (<i>m</i> - <i>p</i> , <i>m</i> - <i>p</i>).
<i>v1t</i>	Holds the matrix of size (<i>q</i> , <i>q</i>).
<i>v2t</i>	Holds the matrix of size (<i>m</i> - <i>q</i> , <i>m</i> - <i>q</i>).
<i>jobsu1</i>	Indicates whether <i>u</i> ₁ is computed. Must be 'Y' or 'O'.
<i>jobsu2</i>	Indicates whether <i>u</i> ₂ is computed. Must be 'Y' or 'O'.
<i>jobv1t</i>	Indicates whether <i>v</i> ₁ ^t is computed. Must be 'Y' or 'O'.
<i>jobv2t</i>	Indicates whether <i>v</i> ₂ ^t is computed. Must be 'Y' or 'O'.
<i>trans</i>	Must be 'N' or 'T'.

signs Must be 'O' or 'D'.

See Also

?bbcisd
xerbla

?orcsd2by1/?uncsd2by1

Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

Syntax

FORTRAN 77:

```
call sorcsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1,
ldu1, u2, ldu2, v1t, ldv1t, work, lwork, iwork, info )
call dorcsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1,
ldu1, u2, ldu2, v1t, ldv1t, work, lwork, iwork, info )
call cuncsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1,
ldu1, u2, ldu2, v1t, ldv1t, work, lwork, rwork, lrwork, iwork, info )
call zuncsd2by1( jobu1, jobu2, jobv1t, m, p, q, x11, ldx11, x21, ldx21, theta, u1,
ldu1, u2, ldu2, v1t, ldv1t, work, lwork, rwork, lrwork, iwork, info )
```

Fortran 95:

```
call orcsd2by1( x11,x21,theta,u1,u2,v1t[,jobu1][,jobu2][,jobv1t][,info] )
call uncisd2by1( x11,x21,theta,u1,u2,v1t[,jobu1][,jobu2][,jobv1t][,info] )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routines ?orcsd2by1/?uncsd2by1 compute the CS decomposition of an m -by- q matrix X with orthonormal columns that has been partitioned into a 2-by-1 block structure:

$$X = \begin{bmatrix} X_{11} \\ X_{21} \end{bmatrix} = \left[\begin{array}{c|c} U_1 & U_2 \end{array} \right] \begin{bmatrix} I & 0 & 0 \\ 0 & C & 0 \\ 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & S & 0 \\ 0 & 0 & I \end{bmatrix} V_1^H$$

X_{11} is p -by- q . The orthogonal/unitary matrices U_1 , U_2 , V_1 , and V_2 are p -by- p , $(m-p)$ -by- $(m-p)$, q -by- q , $(m-q)$ -by- $(m-q)$, respectively. C and S are r -by- r nonnegative diagonal matrices satisfying $C^2 + S^2 = I$, in which $r = \min(p, m-p, q, m-q)$.

Input Parameters

<i>jobu1</i>	CHARACTER. If equal to 'Y', then u_1 is computed. Otherwise, u_1 is not computed.
<i>jobu2</i>	CHARACTER. If equal to 'Y', then u_2 is computed. Otherwise, u_2 is not computed.
<i>jobv1t</i>	CHARACTER. If equal to 'Y', then v_1^t is computed. Otherwise, v_1^t is not computed.
<i>m</i>	INTEGER. The number of rows and columns of the matrix X .
<i>p</i>	INTEGER. The number of rows in x_{11} . $0 \leq p \leq m$.
<i>q</i>	INTEGER. The number of columns in x_{11} . $0 \leq q \leq m$.
<i>x11</i>	REAL for <i>sorcsd2by1</i> DOUBLE PRECISION for <i>dorcsd2by1</i> COMPLEX for <i>cuncsd2by1</i> DOUBLE COMPLEX for <i>zuncsd2by1</i> Array, size (<i>lidx11,q</i>). On entry, the part of the orthogonal matrix whose CSD is desired.
<i>lidx11</i>	INTEGER. The leading dimension of the array x_{11} . $lidx11 \geq \max(1, p)$.
<i>x21</i>	REAL for <i>sorcsd2by1</i> DOUBLE PRECISION for <i>dorcsd2by1</i> COMPLEX for <i>cuncsd2by1</i> DOUBLE COMPLEX for <i>zuncsd2by1</i> Array, size (<i>lidx21,q</i>). On entry, the part of the orthogonal matrix whose CSD is desired.
<i>lidx21</i>	INTEGER. The leading dimension of the array X . $lidx21 \geq \max(1, m - p)$.
<i>ldu1</i>	INTEGER. The leading dimension of the array u_1 . If $jobu1 = 'Y'$, $ldu1 \geq \max(1, p)$.
<i>ldu2</i>	INTEGER. The leading dimension of the array u_2 . If $jobu2 = 'Y'$, $ldu2 \geq \max(1, m-p)$.
<i>ldv1t</i>	INTEGER. The leading dimension of the array v_1^t . If $jobv1t = 'Y'$, $ldv1t \geq \max(1, q)$.
<i>work</i>	REAL for <i>sorcsd2by1</i> DOUBLE PRECISION for <i>dorcsd2by1</i> COMPLEX for <i>cuncsd2by1</i> DOUBLE COMPLEX for <i>zuncsd2by1</i>

Workspace array, size ($\max(1, lwork)$).

lwork

INTEGER. The size of the *work* array. Constraints:

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by *xerbla*.

rwork

REAL for *cuncsd2by1*

DOUBLE PRECISION for *zuncsd2by1*

Workspace array, size ($\max(1, lrwork)$).

lrwork

INTEGER. The size of the *rwork* array. Constraints:

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *rwork* array, returns this value as the first entry of the *rwork* array, and no error message related to *lrwork* is issued by *xerbla*.

iwork

INTEGER. Workspace array, dimension $m - \min(p, m - p, q, m - q)$.

Output Parameters

theta

REAL for *sorcsd2by1*

DOUBLE PRECISION for *dorcsd2by1*

COMPLEX for *cuncsd2by1*

DOUBLE COMPLEX for *zuncsd2by1*

Array, size (r), in which $r = \min(p, m-p, q, m-q)$.

$C = \text{diag}(\cos(\theta(1)), \dots, \cos(\theta(r)))$, and

$S = \text{diag}(\sin(\theta(1)), \dots, \sin(\theta(r)))$.

u1

REAL for *sorcsd2by1*

DOUBLE PRECISION for *dorcsd2by1*

COMPLEX for *cuncsd2by1*

DOUBLE COMPLEX for *zuncsd2by1*

Array, size ($ldu1, p$).

If $\text{jobu1} = 'Y'$, *u1* contains the p -by- p orthogonal/unitary matrix U_1 .

u2

REAL for *sorcsd2by1*

DOUBLE PRECISION for *dorcsd2by1*

COMPLEX for *cuncsd2by1*

DOUBLE COMPLEX for *zuncsd2by1*

Array, size ($ldu2, m-p$).

If $\text{jobu2} = 'Y'$, *u2* contains the $(m-p)$ -by- $(m-p)$ orthogonal/unitary matrix U_2 .

<code>v1t</code>	REAL for <code>sorcsd2by1</code> DOUBLE PRECISION for <code>dorcsd2by1</code> COMPLEX for <code>cuncsd2by1</code> DOUBLE COMPLEX for <code>zuncsd2by1</code> Array, size (<code>ldv1t,q</code>). If <code>jobv1t = 'Y'</code> , <code>v1t</code> contains the q -by- q orthogonal matrix v_1^T or unitary matrix v_1^H .
<code>work</code>	On exit, If <code>info = 0</code> , <code>work(1)</code> returns the optimal <code>lwork</code> . If <code>info > 0</code> , <code>work(2:r)</code> contains the values <code>phi(1), ..., phi(r-1)</code> that, together with <code>theta(1), ..., theta(r)</code> define the matrix in intermediate bidiagonal-block form remaining after nonconvergence. <code>info</code> specifies the number of nonzero <code>phi</code> 's.
<code>rwork</code>	On exit, If <code>info = 0</code> , <code>rwork(1)</code> returns the optimal <code>lrwork</code> . If <code>info > 0</code> , <code>rwork(2:r)</code> contains the values <code>phi(1), ..., phi(r-1)</code> that, together with <code>theta(1), ..., theta(r)</code> define the matrix in intermediate bidiagonal-block form remaining after nonconvergence. <code>info</code> specifies the number of nonzero <code>phi</code> 's.
<code>info</code>	INTEGER. = 0: successful exit < 0: if <code>info = -i</code> , the i -th argument has an illegal value > 0: <code>?orcsd2by1/?uncsd2by1</code> did not converge. See the description of <code>work</code> above for details.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `?orcsd2by1/?orcsd2by1` interface are as follows:

<code>x11</code>	Holds the block of matrix X_{11} of size (p, q) .
<code>x21</code>	Holds the block of matrix X_{21} of size $(m-p, q)$.
<code>theta</code>	Holds the vector of length $r = \min(p, m-p, q, m-q)$.
<code>u1</code>	Holds the matrix u_1 of size (p, p) .
<code>u2</code>	Holds the matrix u_2 of size $(m-p, m-p)$.

v_1^t	Holds the matrix v_1^T or v_1^H of size (q,q) .
$jobu1$	Indicates whether u_1 is computed. Must be ' Y ' or ' O '.
$jobu2$	Indicates whether u_2 is computed. Must be ' Y ' or ' O '.
$jobv1t$	Indicates whether v_1^t is computed. Must be ' Y ' or ' O '.

See Also

?bbcisd
xerbla

Generalized Symmetric Definite Eigenproblems

This section describes LAPACK driver routines used for solving generalized symmetric definite eigenproblems. See also [computational routines](#) that can be called to solve these problems. [Table "Driver Routines for Solving Generalized Symmetric Definite Eigenproblems"](#) lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Generalized Symmetric Definite Eigenproblems

Routine Name	Operation performed
sygv/hegv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem.
sygvd/hegvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.
sygvx/hegvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem.
spgv/hpgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage.
spgvd/hpgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.
spgvx/hpgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage.
sbgv/hbgv	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices.
sbgvd/hbgvd	Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.
sbgvx/hbgvx	Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices.

?sygv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

FORTRAN 77:

```
call ssygv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
call dsygv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, info)
```

Fortran 95:

```
call sygv(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_<?>sygv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, <datatype>* a, lapack_int lda, <datatype>* b, lapack_int ldb,
<datatype>* w );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be symmetric and B is also positive definite.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda^*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda^*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda^*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>a</i> , <i>b</i> , <i>work</i>	REAL for ssygv DOUBLE PRECISION for dsygv.

Arrays:

$a(lda, *)$ contains the upper or lower triangle of the symmetric matrix A , as specified by $uplo$.

The second dimension of a must be at least $\max(1, n)$.

$b(ldb, *)$ contains the upper or lower triangle of the symmetric positive definite matrix B , as specified by $uplo$.

The second dimension of b must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The leading dimension of a ; at least $\max(1, n)$.

ldb

INTEGER. The leading dimension of b ; at least $\max(1, n)$.

$lwork$

INTEGER.

The dimension of the array $work$;

$lwork \geq \max(1, 3n-1)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

Output Parameters

a

On exit, if $jobz = 'V'$, then if $info = 0$, a contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if $itype = 1$ or 2 , $Z^T * B * Z = I$;

if $itype = 3$, $Z^T * \text{inv}(B) * Z = I$;

If $jobz = 'N'$, then on exit the upper triangle (if $uplo = 'U'$) or the lower triangle (if $uplo = 'L'$) of A , including the diagonal, is destroyed.

b

On exit, if $info \leq n$, the part of b containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$.

w

REAL for ssygv

DOUBLE PRECISION for dsygv.

Array, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues in ascending order.

$work(1)$

On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument had an illegal value.

If $info > 0$, [spotrf/dpotrf](#) and [ssyev/dsyev](#) returned an error code:

If $\text{info} = i \leq n$, `ssyev/dsyev` failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If $\text{info} = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sygv` interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size (n, n) .
<i>w</i>	Holds the vector of length n .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>jobz</i>	Must be ' <code>N</code> ' or ' <code>V</code> '. The default value is ' <code>N</code> '.
<i>uplo</i>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

For optimum performance use $\text{lwork} \geq (nb+2)*n$, where nb is the blocksize for `ssytrd/dsytrd` returned by `ilaenv`.

If it is not clear how much workspace to supply, use a generous value of `lwork` (or `liwork`) for the first run or set `lwork = -1` (`liwork = -1`).

If `lwork` (or `liwork`) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work, iwork`) on exit. Use this value (`work(1), iwork(1)`) for subsequent runs.

If `lwork = -1` (`liwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work, iwork`). This operation is called a workspace query.

Note that if `work` (`liwork`) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax

FORTRAN 77:

```
call chegv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info)
call zhegv(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, info)
```

Fortran 95:

```
call hegv(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_chevg( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b,
lapack_int ldb, float* w );

lapack_int LAPACKE_zhegv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b,
lapack_int ldb, double* w );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda B^*x, \quad A^*B^*x = \lambda x, \text{ or } B^*A^*x = \lambda x.$$

Here A and B are assumed to be Hermitian and B is also positive definite.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

itype INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

if *itype* = 1, the problem type is $A^*x = \lambda B^*x$;
 if *itype* = 2, the problem type is $A^*B^*x = \lambda x$;
 if *itype* = 3, the problem type is $B^*A^*x = \lambda x$.

jobz CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.
 If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *a* and *b* store the upper triangles of A and B ;
 If *uplo* = 'L', arrays *a* and *b* store the lower triangles of A and B .

n INTEGER. The order of the matrices A and B ($n \geq 0$).

a, *b*, *work* COMPLEX for chegv

DOUBLE COMPLEX for zhegv.

Arrays:

a(*lda*,*) contains the upper or lower triangle of the Hermitian matrix A , as specified by *uplo*.

The second dimension of *a* must be at least $\max(1, n)$.

$b(lDb, *)$ contains the upper or lower triangle of the Hermitian positive definite matrix B , as specified by $uplo$.

The second dimension of b must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, n)$.

ldb INTEGER. The leading dimension of b ; at least $\max(1, n)$.

$lwork$ INTEGER.

The dimension of the array $work$; $lwork \geq \max(1, 2n-1)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See [Application Notes](#) for the suggested value of $lwork$.

$rwork$ REAL for [chegv](#)

DOUBLE PRECISION for [zhgsv](#).

Workspace array, DIMENSION at least $\max(1, 3n-2)$.

Output Parameters

a On exit, if $jobz = 'V'$, then if $info = 0$, a contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

```
if  $itype = 1$  or  $2$ ,  $Z^H * B * Z = I$ ;  
if  $itype = 3$ ,  $Z^H * \text{inv}(B) * Z = I$ ;
```

If $jobz = 'N'$, then on exit the upper triangle (if $uplo = 'U'$) or the lower triangle (if $uplo = 'L'$) of A , including the diagonal, is destroyed.

b On exit, if $info \leq n$, the part of b containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.

w REAL for [chegv](#)

DOUBLE PRECISION for [zhgsv](#).

Array, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues in ascending order.

$work(1)$ On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

$info$ INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument has an illegal value.

If $info > 0$, [cpotrf/zpotrf](#) and [cheev/zheev](#) return an error code:

If $\text{info} = i \leq n$, `cheev/zheev` fails to converge, and i off-diagonal elements of an intermediate tridiagonal do not converge to zero;

If $\text{info} = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B can not be completed and no eigenvalues or eigenvectors are computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hegv` interface are the following:

a	Holds the matrix A of size (n, n) .
b	Holds the matrix B of size (n, n) .
w	Holds the vector of length n .
$itype$	Must be 1, 2, or 3. The default value is 1.
$jobz$	Must be ' <code>N</code> ' or ' <code>V</code> '. The default value is ' <code>N</code> '.
$uplo$	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

For optimum performance use $lwork \geq (nb+1)*n$, where nb is the blocksize for `chetrd/zhetrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of $lwork$ for the first run or set $lwork = -1$.

If you choose the first option and set any of admissible $lwork$ sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array `work` on exit. Use this value (`work(1)`) for subsequent runs.

If you set $lwork = -1$, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`). This operation is called a workspace query.

Note that if you set $lwork$ to less than the minimal required value and not -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sygvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

FORTRAN 77:

```
call ssygvd( $itype$ ,  $jobz$ ,  $uplo$ ,  $n$ ,  $a$ ,  $lda$ ,  $b$ ,  $ldb$ ,  $w$ ,  $work$ ,  $lwork$ ,  $iwork$ ,  $liwork$ ,  $info$ )
call dsygvd( $itype$ ,  $jobz$ ,  $uplo$ ,  $n$ ,  $a$ ,  $lda$ ,  $b$ ,  $ldb$ ,  $w$ ,  $work$ ,  $lwork$ ,  $iwork$ ,  $liwork$ ,  $info$ )
```

Fortran 95:

```
call sygvd( $a$ ,  $b$ ,  $w$  [ $,itype$ ] [ $,jobz$ ] [ $,uplo$ ] [ $,info$ ])
```

C:

```
lapack_int LAPACKE_<?>sygvd( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, <datatype>* a, lapack_int lda, <datatype>* b, lapack_int ldb,
<datatype>* w );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda*B^*x, \quad A*B^*x = \lambda*x, \quad \text{or} \quad B^*A^*x = \lambda*x.$$

Here A and B are assumed to be symmetric and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda*B^*x$; if <i>itype</i> = 2, the problem type is $A*B^*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>a</i> , <i>b</i> , <i>work</i>	REAL for ssygvd DOUBLE PRECISION for dsygvd. Arrays: $a(lda,*)$ contains the upper or lower triangle of the symmetric matrix A , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. $b ldb,*)$ contains the upper or lower triangle of the symmetric positive definite matrix B , as specified by <i>uplo</i> . The second dimension of <i>b</i> must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

ldb INTEGER. The leading dimension of *b*; at least $\max(1, n)$.

lwork INTEGER.

The dimension of the array *work*.

Constraints:

If $n \leq 1$, *lwork* ≥ 1 ;

If *jobz* = 'N' and $n > 1$, *lwork* $< 2n+1$;

If *jobz* = 'V' and $n > 1$, *lwork* $< 2n^2+6n+1$.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork INTEGER.

Workspace array, its dimension $\max(1, lwork)$.

liwork INTEGER.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1$, *liwork* ≥ 1 ;

If *jobz* = 'N' and $n > 1$, *liwork* ≥ 1 ;

If *jobz* = 'V' and $n > 1$, *liwork* $\geq 5n+3$.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

a On exit, if *jobz* = 'V', then if *info* = 0, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^T * B * Z = I$;

if *itype* = 3, $Z^T * \text{inv}(B) * Z = I$;

If *jobz* = 'N', then on exit the upper triangle (if *uplo* = 'U') or the lower triangle (if *uplo* = 'L') of *A*, including the diagonal, is destroyed.

b On exit, if *info* $\leq n$, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$.

w REAL for ssygvd

DOUBLE PRECISION for `dsygvd`.

`Array`, DIMENSION at least `max(1, n)`.

If `info = 0`, contains the eigenvalues in ascending order.

`work(1)`
On exit, if `info = 0`, then `work(1)` returns the required minimal size of `lwork`.

`iwork(1)`
On exit, if `info = 0`, then `iwork(1)` returns the required minimal size of `liwork`.

`info` INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the i -th argument had an illegal value.

If `info > 0`, an error code is returned as specified below.

- For $info \leq N$:

- If `info = i`, with $i \leq n$, and `jobz = 'N'`, then the algorithm failed to converge; i off-diagonal elements of an intermediate tridiagonal form did not converge to zero.
- If `jobz = 'V'`, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $info/(n+1)$ through $\text{mod}(info, n+1)$.

- For $info > N$:

- If `info = n + i`, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sygvd` interface are the following:

<code>a</code>	Holds the matrix A of size (n, n) .
<code>b</code>	Holds the matrix B of size (n, n) .
<code>w</code>	Holds the vector of length n .
<code>itype</code>	Must be 1, 2, or 3. The default value is 1.
<code>jobz</code>	Must be ' <code>N</code> ' or ' <code>V</code> '. The default value is ' <code>N</code> '.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.

Application Notes

If it is not clear how much workspace to supply, use a generous value of `lwork` (or `liwork`) for the first run or set `lwork = -1` (`liwork = -1`).

If $lwork$ (or $liwork$) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$) on exit. Use this value ($work(1)$, $iwork(1)$) for subsequent runs.

If $lwork = -1$ ($liwork = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array ($work$, $iwork$). This operation is called a workspace query.

Note that if $work$ ($liwork$) is less than the minimal required value and is not equal to -1 , the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

FORTRAN 77:

```
call chegvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, lrwork,  
iwork, liwork, info)  
call zhegvd(itype, jobz, uplo, n, a, lda, b, ldb, w, work, lwork, rwork, lrwork,  
iwork, liwork, info)
```

Fortran 95:

```
call hegvd(a, b, w [,itype] [,jobz] [,uplo] [,info])
```

C:

```
lapack_int LAPACKE_chegvd( int matrix_layout, lapack_int itype, char jobz, char uplo,  
lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b,  
lapack_int ldb, float* w );  
lapack_int LAPACKE_zhegvd( int matrix_layout, lapack_int itype, char jobz, char uplo,  
lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b,  
lapack_int ldb, double* w );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda B^*x, \quad A^*B^*x = \lambda x, \text{ or } B^*A^*x = \lambda x.$$

Here A and B are assumed to be Hermitian and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A*x = \lambda*B*x$; if <i>itype</i> = 2, the problem type is $A*B*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B*A*x = \lambda*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> ($n \geq 0$).
<i>a</i> , <i>b</i> , <i>work</i>	COMPLEX for <code>chegvd</code> DOUBLE COMPLEX for <code>zhegvd</code> . Arrays: <i>a</i> (<i>lda</i> ,*) contains the upper or lower triangle of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb</i> ,*) contains the upper or lower triangle of the Hermitian positive definite matrix <i>B</i> , as specified by <i>uplo</i> . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$. <i>lda</i> INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$. <i>ldb</i> INTEGER. The leading dimension of <i>b</i> ; at least $\max(1, n)$. <i>lwork</i> INTEGER. The dimension of the array <i>work</i> . Constraints: If $n \leq 1$, <i>lwork</i> ≥ 1 ; If <i>jobz</i> = 'N' and $n > 1$, <i>lwork</i> $\geq n+1$; If <i>jobz</i> = 'V' and $n > 1$, <i>lwork</i> $\geq n^2+2n$. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> , <i>rwork</i> and <i>iwork</i> arrays, returns these values as the first entries of the <i>work</i> , <i>rwork</i> and <i>iwork</i> arrays, and no error message related to <i>lwork</i> or <i>lwork</i> or <i>liwork</i> is issued by <code>xerbla</code> . See <i>Application Notes</i> for details.
<i>rwork</i>	REAL for <code>chegvd</code> DOUBLE PRECISION for <code>zhegvd</code> .

Workspace array, DIMENSION max(1, lrwork).

lrwork

INTEGER.

The dimension of the array *rwork*.

Constraints:

If $n \leq 1$, $lrwork \geq 1$;

If $jobz = 'N'$ and $n > 1$, $lrwork \geq n$;

If $jobz = 'V'$ and $n > 1$, $lrwork \geq 2n^2 + 5n + 1$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork

INTEGER.

Workspace array, DIMENSION max(1, liwork).

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1$, $liwork \geq 1$;

If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;

If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n + 3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work*, *rwork* and *iwork* arrays, returns these values as the first entries of the *work*, *rwork* and *iwork* arrays, and no error message related to *lwork* or *lrwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

a

On exit, if $jobz = 'V'$, then if $info = 0$, *a* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if $itype = 1$ or 2 , $Z^H * B * Z = I$;

if $itype = 3$, $Z^H * \text{inv}(B) * Z = I$;

If $jobz = 'N'$, then on exit the upper triangle (if $uplo = 'U'$) or the lower triangle (if $uplo = 'L'$) of *A*, including the diagonal, is destroyed.

b

On exit, if $info \leq n$, the part of *b* containing the matrix is overwritten by the triangular factor *U* or *L* from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$.

w

REAL for [chegvd](#)

DOUBLE PRECISION for [zhegvd](#).

Array, DIMENSION at least $\max(1, n)$.

If $\text{info} = 0$, contains the eigenvalues in ascending order.

$\text{work}(1)$ On exit, if $\text{info} = 0$, then $\text{work}(1)$ returns the required minimal size of lwork .

$\text{rwork}(1)$ On exit, if $\text{info} = 0$, then $\text{rwork}(1)$ returns the required minimal size of lrwork .

$\text{iwork}(1)$ On exit, if $\text{info} = 0$, then $\text{iwork}(1)$ returns the required minimal size of liwork .

info INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th argument had an illegal value.

If $\text{info} = i$, and $\text{jobz} = \text{'N'}$, then the algorithm failed to converge; i off-diagonal elements of an intermediate tridiagonal form did not converge to zero;

If $\text{info} = i$, and $\text{jobz} = \text{'V'}$, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $\text{info}/(n+1)$ through $\text{mod}(\text{info}, n+1)$.

If $\text{info} = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hegvd` interface are the following:

a Holds the matrix A of size (n, n) .

b Holds the matrix B of size (n, n) .

w Holds the vector of length n .

$itype$ Must be 1, 2, or 3. The default value is 1.

$jobz$ Must be ' N ' or ' V '. The default value is ' N '.

$uplo$ Must be ' U ' or ' L '. The default value is ' U '.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of lwork (liwork or lrwork) for the first run or set $\text{lwork} = -1$ ($\text{liwork} = -1$, $\text{lrwork} = -1$).

If you choose the first option and set any of admissible lwork (liwork or lrwork) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (work , iwork , rwork) on exit. Use this value ($\text{work}(1)$, $\text{iwork}(1)$, $\text{rwork}(1)$) for subsequent runs.

If you set `lwork = -1` (`liwork = -1`, `lrwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`). This operation is called a workspace query.

Note that if you set `lwork` (`liwork`, `lrwork`) to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

FORTRAN 77:

```
call ssygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m,
w, z, ldz, work, lwork, iwork, ifail, info)
call dsygvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m,
w, z, ldz, work, lwork, iwork, ifail, info)
```

Fortran 95:

```
call sygvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol]
[,info])
```

C:

```
lapack_int LAPACKE_<?>sygvx( int matrix_layout, lapack_int itype, char jobz, char
range, char uplo, lapack_int n, <datatype>* a, lapack_int lda, <datatype>* b,
lapack_int ldb, <datatype> vl, <datatype> vu, lapack_int il, lapack_int iu, <datatype>
abstol, lapack_int* m, <datatype>* w, <datatype>* z, lapack_int ldz, lapack_int*
ifail );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be symmetric and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>itype</code>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <code>itype = 1</code> , the problem type is $A^*x = \lambda^*B^*x$;
--------------------	--

if $itype = 2$, the problem type is $A^*B^*x = \lambda x$;
 if $itype = 3$, the problem type is $B^*A^*x = \lambda x$.

jobz

CHARACTER*1. Must be 'N' or 'V'.
If $jobz = 'N'$, then compute eigenvalues only.
If $jobz = 'V'$, then compute eigenvalues and eigenvectors.

range

CHARACTER*1. Must be 'A' or 'V' or 'I'.
If $range = 'A'$, the routine computes all eigenvalues.
If $range = 'V'$, the routine computes eigenvalues $\lambda(i)$ in the half-open interval:
 $vl < \lambda(i) \leq vu$.
If $range = 'I'$, the routine computes eigenvalues with indices il to iu .

uplo

CHARACTER*1. Must be 'U' or 'L'.
If $uplo = 'U'$, arrays a and b store the upper triangles of A and B ;
If $uplo = 'L'$, arrays a and b store the lower triangles of A and B .

n

INTEGER. The order of the matrices A and B ($n \geq 0$).

a, b, work

REAL for ssygvx
 DOUBLE PRECISION for dsygvx.
Arrays:

$a(lda, *)$ contains the upper or lower triangle of the symmetric matrix A , as specified by *uplo*.

The second dimension of a must be at least $\max(1, n)$.

$b(ldb, *)$ contains the upper or lower triangle of the symmetric positive definite matrix B , as specified by *uplo*.

The second dimension of b must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, lwork)$.

lda

INTEGER. The leading dimension of a ; at least $\max(1, n)$.

ldb

INTEGER. The leading dimension of b ; at least $\max(1, n)$.

vl, vu

REAL for ssygvx
 DOUBLE PRECISION for dsygvx.

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If $range = 'A'$ or ' I' , vl and vu are not referenced.

il, iu

INTEGER.

If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$
if $n = 0$.

If $range = 'A'$ or ' V ', il and iu are not referenced.

$abstol$

REAL for ssygvx

DOUBLE PRECISION for dsygvx. The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array z . Constraints:

$ldz \geq 1$; if $jobz = 'V'$, $ldz \geq \max(1, n)$.

$lwork$

INTEGER.

The dimension of the array $work$;

$lwork < \max(1, 8n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

See *Application Notes* for the suggested value of $lwork$.

$iwork$

INTEGER.

Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

a

On exit, the upper triangle (if $uplo = 'U'$) or the lower triangle (if $uplo = 'L'$) of A , including the diagonal, is overwritten.

b

On exit, if $info \leq n$, the part of b containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$.

m

INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$. If $range = 'A'$, $m = n$, and if $range = 'I'$,
 $m = iu - il + 1$.

w, z

REAL for ssygvx

DOUBLE PRECISION for dsygvx.

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$.

The first m elements of w contain the selected eigenvalues in ascending order.

$z(ldz,*)$.

The second dimension of z must be at least $\max(1, m)$.

If $jobz = 'V'$, then if $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized as follows:

```
if itype = 1 or 2,  $Z^T * B * Z = I$ ;  
if itype = 3,  $Z^T * \text{inv}(B) * Z = I$ ;
```

If $jobz = 'N'$, then z is not referenced.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $range = 'V'$, the exact value of m is not known in advance and an upper bound must be used.

work(1)

On exit, if $info = 0$, then *work(1)* returns the required minimal size of *lwork*.

ifail

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of *ifail* are zero; if $info > 0$, the *ifail* contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then *ifail* is not referenced.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i th argument had an illegal value.

If $info > 0$, *spotrf/dpotrf* and *ssyevx/dsyevx* returned an error code:

If $info = i \leq n$, *ssyevx/dsyevx* failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *sygvx* interface are the following:

<i>a</i>	Holds the matrix A of size (n, n) .
<i>b</i>	Holds the matrix B of size (n, n) .
<i>w</i>	Holds the vector of length n .
<i>z</i>	Holds the matrix Z of size (n, n) , where the values n and m are significant.

<i>ifail</i>	Holds the vector of length n .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be ' <i>U</i> ' or ' <i>L</i> '. The default value is ' <i>U</i> '.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_{\text{WP}}$.
<i>jobz</i>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted. Note that there will be an error condition if <i>ifail</i> is present and z is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present. Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + \varepsilon * \max(|a|, |b|)$, where ε is the machine precision.

If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ is used as tolerance, where T is the tridiagonal matrix obtained by reducing C to tridiagonal form, where C is the symmetric matrix of the standard symmetric problem to which the generalized problem is transformed. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * ?\text{lamch}('S')$, not zero.

If this routine returns with *info* > 0 , indicating that some eigenvectors did not converge, set *abstol* to $2 * ?\text{lamch}('S')$.

For optimum performance use *lwork* $\geq (nb+3)*n$, where *nb* is the blocksize for *ssytrd/dsytrd* returned by *ilaenv*.

If it is not clear how much workspace to supply, use a generous value of *lwork* for the first run, or set *lwork* = -1.

In first case the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If *lwork* = -1, then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if *lwork* is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hegvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax**FORTRAN 77:**

```
call chegvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m,
w, z, ldz, work, lwork, rwork, iwork, ifail, info)
call zhegvx(itype, jobz, range, uplo, n, a, lda, b, ldb, vl, vu, il, iu, abstol, m,
w, z, ldz, work, lwork, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hegvx(a, b, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,abstol]
[,info])
```

C:

```
lapack_int LAPACKE_chegvx( int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float*
b, lapack_int ldb, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz, lapack_int* ifail );
lapack_int LAPACKE_zhegvx( int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, lapack_complex_double* a, lapack_int lda,
lapack_complex_double* b, lapack_int ldb, double vl, double vu, lapack_int il,
lapack_int iu, double abstol, lapack_int* m, double* w, lapack_complex_double* z,
lapack_int ldz, lapack_int* ifail );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda * B^*x, \quad A^*B^*x = \lambda * x, \text{ or } B^*A^*x = \lambda * x.$$

Here A and B are assumed to be Hermitian and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if $itype = 1$, the problem type is $A^*x = \lambda * B^*x$; if $itype = 2$, the problem type is $A^*B^*x = \lambda * x$; if $itype = 3$, the problem type is $B^*A^*x = \lambda * x$.
-------	--

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues <i>lambda(i)</i> in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> (<i>n</i> \geq 0).
<i>a, b, work</i>	COMPLEX for <i>chegvx</i> DOUBLE COMPLEX for <i>zhegvx</i> . Arrays: <i>a</i> (<i>lda,*</i>) contains the upper or lower triangle of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>b</i> (<i>ldb,*</i>) contains the upper or lower triangle of the Hermitian positive definite matrix <i>B</i> , as specified by <i>uplo</i> . The second dimension of <i>b</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, n)$.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; at least $\max(1, n)$.
<i>vl, vu</i>	REAL for <i>chegvx</i> DOUBLE PRECISION for <i>zhegvx</i> . If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il, iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if <i>n</i> > 0; <i>il=1</i> and <i>iu=0</i> if <i>n</i> = 0.

If `range = 'A'` or `'V'`, `il` and `iu` are not referenced.

`abstol`

REAL for `chegvx`

DOUBLE PRECISION for `zhegvx`.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

`ldz`

INTEGER. The leading dimension of the output array `z`. Constraints:

`ldz ≥ 1`; if `jobz = 'V'`, `ldz ≥ max(1, n)`.

`lwork`

INTEGER.

The dimension of the array `work`; `lwork ≥ max(1, 2n)`.

If `lwork = -1`, then a workspace query is assumed; the routine only calculates the optimal size of the `work` array, returns this value as the first entry of the `work` array, and no error message related to `lwork` is issued by `xerbla`.

See *Application Notes* for the suggested value of `lwork`.

`rwork`

REAL for `chegvx`

DOUBLE PRECISION for `zhegvx`.

Workspace array, DIMENSION at least `max(1, 7n)`.

`iwork`

INTEGER.

Workspace array, DIMENSION at least `max(1, 5n)`.

Output Parameters

`a`

On exit, the upper triangle (if `uplo = 'U'`) or the lower triangle (if `uplo = 'L'`) of `A`, including the diagonal, is overwritten.

`b`

On exit, if `info ≤ n`, the part of `b` containing the matrix is overwritten by the triangular factor `U` or `L` from the Cholesky factorization $B = U^H * U$ or $B = L^H * L$.

`m`

INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$. If `range = 'A'`, $m = n$, and if `range = 'I'`,

$m = iu - il + 1$.

`w`

REAL for `chegvx`

DOUBLE PRECISION for `zhegvx`.

Array, DIMENSION at least `max(1, n)`.

The first `m` elements of `w` contain the selected eigenvalues in ascending order.

`z`

COMPLEX for `chegvx`

DOUBLE COMPLEX for `zhegvx`.

Array `z(ldz, *)`. The second dimension of `z` must be at least `max(1, m)`.

If $\text{jobz} = \text{'V'}$, then if $\text{info} = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized as follows:

```
if  $\text{itype} = 1$  or  $2$ ,  $Z^H * B * Z = I$ ;  
if  $\text{itype} = 3$ ,  $Z^H * \text{inv}(B) * Z = I$ ;
```

If $\text{jobz} = \text{'N'}$, then z is not referenced.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in ifail .

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array z ; if $\text{range} = \text{'V'}$, the exact value of m is not known in advance and an upper bound must be used.

$\text{work}(1)$

On exit, if $\text{info} = 0$, then $\text{work}(1)$ returns the required minimal size of lwork .

ifail

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $\text{jobz} = \text{'V'}$, then if $\text{info} = 0$, the first m elements of ifail are zero; if $\text{info} > 0$, the ifail contains the indices of the eigenvectors that failed to converge.

If $\text{jobz} = \text{'N'}$, then ifail is not referenced.

info

INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i th argument had an illegal value.

If $\text{info} > 0$, cpotrf/zpotrf and cheevx/zheevx returned an error code:

If $\text{info} = i \leq n$, cheevx/zheevx failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array ifail ;

If $\text{info} = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine hegvx interface are the following:

a	Holds the matrix A of size (n, n) .
b	Holds the matrix B of size (n, n) .
w	Holds the vector of length n .
z	Holds the matrix Z of size (n, n) , where the values n and m are significant.

<i>ifail</i>	Holds the vector of length n .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_{\text{WP}}$.
<i>jobz</i>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted. Note that there will be an error condition if <i>ifail</i> is present and z is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present. Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * ||T||_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing C to tridiagonal form, where C is the symmetric matrix of the standard symmetric problem to which the generalized problem is transformed. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * ?\text{lamch}('S')$, not zero.

If this routine returns with *info* > 0 , indicating that some eigenvectors did not converge, try setting *abstol* to $2 * ?\text{lamch}('S')$.

For optimum performance use *lwork* $\geq (nb+1)*n$, where *nb* is the blocksize for `chetrd/zhetrd` returned by `ilaenv`.

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

FORTRAN 77:

```
call sspgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info)
call dspgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, info)
```

Fortran 95:

```
call spgv(ap, bp, w [,itype] [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_<?>spgv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, <datatype>* ap, <datatype>* bp, <datatype>* w, <datatype>* z, lapack_int
ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda*B^*x, A^*B^*x = \lambda*x, \text{ or } B^*A^*x = \lambda*x.$$

Here *A* and *B* are assumed to be symmetric, stored in packed format, and *B* is also positive definite.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

itype INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

```
if itype = 1, the problem type is A*x = lambda*B*x;
if itype = 2, the problem type is A*B*x = lambda*x;
if itype = 3, the problem type is B*A*x = lambda*x.
```

jobz CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

uplo CHARACTER*1. Must be 'U' or 'L'.

If *uplo* = 'U', arrays *ap* and *bp* store the upper triangles of *A* and *B*;

If $uplo = 'L'$, arrays ap and bp store the lower triangles of A and B .

n

INTEGER. The order of the matrices A and B ($n \geq 0$).

$ap, bp, work$

REAL for sspgv

DOUBLE PRECISION for dspgv.

Arrays:

$ap(*)$ contains the packed upper or lower triangle of the symmetric matrix A , as specified by $uplo$.

The dimension of ap must be at least $\max(1, n*(n+1)/2)$.

$bp(*)$ contains the packed upper or lower triangle of the symmetric matrix B , as specified by $uplo$.

The dimension of bp must be at least $\max(1, n*(n+1)/2)$.

$work(*)$ is a workspace array, DIMENSION at least $\max(1, 3n)$.

ldz

INTEGER. The leading dimension of the output array z ; $ldz \geq 1$. If $jobz = 'V'$, $ldz \geq \max(1, n)$.

Output Parameters

ap

On exit, the contents of ap are overwritten.

bp

On exit, contains the triangular factor U or L from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as B .

w, z

REAL for sspgv

DOUBLE PRECISION for dspgv.

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues in ascending order.

$z(ldz,*)$.

The second dimension of z must be at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:

if $itype = 1$ or 2 , $Z^T * B * Z = I$;

if $itype = 3$, $Z^T * \text{inv}(B) * Z = I$;

If $jobz = 'N'$, then z is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument had an illegal value.

If $info > 0$, spptrf/dpptrf and sspev/dspev returned an error code:

If $info = i \leq n$, sspev/dspev failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spgv` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>bp</code>	Holds the array B of size $(n*(n+1)/2)$.
<code>w</code>	Holds the vector with the number of elements n .
<code>z</code>	Holds the matrix Z of size (n, n) .
<code>itype</code>	Must be 1, 2, or 3. The default value is 1.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>jobz</code>	Restored based on the presence of the argument <code>z</code> as follows: <code>jobz = 'V'</code> , if <code>z</code> is present, <code>jobz = 'N'</code> , if <code>z</code> is omitted.

?hpgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with matrices in packed storage.

Syntax

FORTRAN 77:

```
call chpgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork, info)
call zhpgv(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hpgv(ap, bp, w [,itype] [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_chpgv( int matrix_layout, lapack_int itype, char jobz, char uplo,
                           lapack_int n, lapack_complex_float* ap, lapack_complex_float* bp, float* w,
                           lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhpgv( int matrix_layout, lapack_int itype, char jobz, char uplo,
                           lapack_int n, lapack_complex_double* ap, lapack_complex_double* bp, double* w,
                           lapack_complex_double* z, lapack_int ldz );
```

Include Files

- Fortran: `mkl.fi`

- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda*B^*x, A^*B^*x = \lambda*x, \text{ or } B^*A^*x = \lambda*x.$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ap</i> , <i>bp</i> , <i>work</i>	COMPLEX for chpgv DOUBLE COMPLEX for zhpgv. Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of the Hermitian matrix A , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>bp</i> (*) contains the packed upper or lower triangle of the Hermitian matrix B , as specified by <i>uplo</i> . The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$. <i>work</i> (*) is a workspace array, DIMENSION at least $\max(1, 2n-1)$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; $ldz \geq 1$. If <i>jobz</i> = 'V', $ldz \geq \max(1, n)$.
<i>rwork</i>	REAL for chpgv DOUBLE PRECISION for zhpgv.

Workspace array, DIMENSION at least $\max(1, 3n-2)$.

Output Parameters

<i>ap</i>	On exit, the contents of <i>ap</i> are overwritten.
<i>bp</i>	On exit, contains the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$, in the same storage format as <i>B</i> .
<i>w</i>	REAL for chpgv DOUBLE PRECISION for zhpgv. Array, DIMENSION at least $\max(1, n)$. If <i>info</i> = 0, contains the eigenvalues in ascending order.
<i>z</i>	COMPLEX for chpgv DOUBLE COMPLEX for zhpgv. Array $z(Idz,*)$. The second dimension of <i>z</i> must be at least $\max(1, n)$. If <i>jobz</i> = 'V', then if <i>info</i> = 0, <i>z</i> contains the matrix <i>Z</i> of eigenvectors. The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^H * B * Z = I$; if <i>itype</i> = 3, $Z^H * \text{inv}(B) * Z = I$; If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th argument had an illegal value. If <i>info</i> > 0, cpptrf/zpptrf and chpev/zheev returned an error code: If <i>info</i> = $i \leq n$, chpev/zheev failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero; If <i>info</i> = $n + i$, for $1 \leq i \leq n$, then the leading minor of order <i>i</i> of <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpgv` interface are the following:

<i>ap</i>	Holds the array <i>A</i> of size $(n*(n+1)/2)$.
<i>bp</i>	Holds the array <i>B</i> of size $(n*(n+1)/2)$.
<i>w</i>	Holds the vector with the number of elements <i>n</i> .

<i>z</i>	Holds the matrix Z of size (n, n) .
<i>itype</i>	Must be 1, 2, or 3. The default value is 1.
<i>uplo</i>	Must be ' <i>U</i> ' or ' <i>L</i> '. The default value is ' <i>U</i> '.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = ' <i>V</i> ', if <i>z</i> is present, <i>jobz</i> = ' <i>N</i> ', if <i>z</i> is omitted.

?spgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.

Syntax**FORTRAN 77:**

```
call sspgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, iwork, liwork, info)
call dspgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call spgvd(ap, bp, w [, itype] [, uplo] [, z] [, info])
```

C:

```
lapack_int LAPACKE_<?>spgvd( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, <datatype>* ap, <datatype>* bp, <datatype>* w, <datatype>* z, lapack_int
ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda B^*x, \quad A^*B^*x = \lambda x, \quad \text{or} \quad B^*A^*x = \lambda x.$$

Here *A* and *B* are assumed to be symmetric, stored in packed format, and *B* is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda B^*x$;
--------------	--

if $itype = 2$, the problem type is $A^*B^*x = \lambda x$;
 if $itype = 3$, the problem type is $B^*A^*x = \lambda x$.

jobz

CHARACTER*1. Must be 'N' or 'V'.
If $jobz = 'N'$, then compute eigenvalues only.
If $jobz = 'V'$, then compute eigenvalues and eigenvectors.

uplo

CHARACTER*1. Must be 'U' or 'L'.
If $uplo = 'U'$, arrays *ap* and *bp* store the upper triangles of *A* and *B*;
If $uplo = 'L'$, arrays *ap* and *bp* store the lower triangles of *A* and *B*.

n

INTEGER. The order of the matrices *A* and *B* ($n \geq 0$).

ap, bp, work

REAL **for** sspgvd
 DOUBLE PRECISION **for** dspgvd.

Arrays:

ap(*) contains the packed upper or lower triangle of the symmetric matrix *A*, as specified by *uplo*.

The dimension of *ap* must be at least $\max(1, n*(n+1)/2)$.

bp(*) contains the packed upper or lower triangle of the symmetric matrix *B*, as specified by *uplo*.

The dimension of *bp* must be at least $\max(1, n*(n+1)/2)$.

work is a workspace array, its dimension $\max(1, lwork)$.

ldz

INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If $jobz = 'V'$, $ldz \geq \max(1, n)$.

lwork

INTEGER.

The dimension of the array *work*.

Constraints:

If $n \leq 1$, $lwork \geq 1$;

If $jobz = 'N'$ and $n > 1$, $lwork \geq 2n$;

If $jobz = 'V'$ and $n > 1$, $lwork \geq 2n^2 + 6n + 1$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by [xerbla](#). See *Application Notes* for details.

iwork

INTEGER.

Workspace array, dimension $\max(1, lwork)$.

liwork

INTEGER.

The dimension of the array *iwork*.

Constraints:

If $n \leq 1$, $liwork \geq 1$;
 If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;
 If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n+3$.
 If $liwork = -1$, then a workspace query is assumed; the routine only calculates the required sizes of the *work* and *iwork* arrays, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by *xerbla*. See *Application Notes* for details.

Output Parameters

ap

On exit, the contents of *ap* are overwritten.

bp

On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as *B*.

w, z

REAL for sspgv

DOUBLE PRECISION for dspgv.

Arrays:

w(*), DIMENSION at least max(1, *n*).

If *info* = 0, contains the eigenvalues in ascending order.

z(*ldz*,*).

The second dimension of *z* must be at least max(1, *n*).

If $jobz = 'V'$, then if *info* = 0, *z* contains the matrix *Z* of eigenvectors. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^T * B * Z = I$;

if *itype* = 3, $Z^T * \text{inv}(B) * Z = I$;

If $jobz = 'N'$, then *z* is not referenced.

work(1)

On exit, if *info* = 0, then *work*(1) returns the required minimal size of *lwork*.

iwork(1)

On exit, if *info* = 0, then *iwork*(1) returns the required minimal size of *liwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the *i*-th argument had an illegal value.

If *info* > 0, *spptrf/dpptrf* and *sspevd/dspevd* returned an error code:

If *info* = $i \leq n$, *sspevd/dspevd* failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If *info* = $n + i$, for $1 \leq i \leq n$, then the leading minor of order *i* of *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spgvd` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>bp</code>	Holds the array B of size $(n*(n+1)/2)$.
<code>w</code>	Holds the vector with the number of elements n .
<code>z</code>	Holds the matrix Z of size (n, n) .
<code>itype</code>	Must be 1, 2, or 3. The default value is 1.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>jobz</code>	Restored based on the presence of the argument <code>z</code> as follows: $\text{jobz} = \text{'V}'$, if <code>z</code> is present, $\text{jobz} = \text{'N}'$, if <code>z</code> is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of `lwork` (or `liwork`) for the first run, or set `lwork = -1` (`liwork = -1`).

If `lwork` (or `liwork`) has any of admissible sizes, which is no less than the minimal value described, then the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`) on exit. Use this value (`work(1)`, `iwork(1)`) for subsequent runs.

If `lwork = -1` (`liwork = -1`), then the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`). This operation is called a workspace query.

Note that if `lwork` (`liwork`) is less than the minimal required value and is not equal to -1, then the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hpgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

FORTRAN 77:

```
call chpgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, rwork, lrwork,
iwork, liwork, info)
```

```
call zhpgvd(itype, jobz, uplo, n, ap, bp, w, z, ldz, work, lwork, rwork, lrwork,
iwork, liwork, info)
```

Fortran 95:

```
call hpgvd(ap, bp, w [,itype] [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_chpgvd( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_float* ap, lapack_complex_float* bp, float* w,
lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhpgvd( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_double* ap, lapack_complex_double* bp, double* w,
lapack_complex_double* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda*B^*x, A^*B^*x = \lambda*x, \text{ or } B^*A^*x = \lambda*x.$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>itype</i>	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if <i>itype</i> = 1, the problem type is $A^*x = \lambda*B^*x$; if <i>itype</i> = 2, the problem type is $A^*B^*x = \lambda*x$; if <i>itype</i> = 3, the problem type is $B^*A^*x = \lambda*x$.
<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ap</i> , <i>bp</i> , <i>work</i>	COMPLEX for chpgvd DOUBLE COMPLEX for zhpgvd. Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of the Hermitian matrix A , as specified by <i>uplo</i> .

The dimension of ap must be at least $\max(1, n*(n+1)/2)$.

$bp(*)$ contains the packed upper or lower triangle of the Hermitian matrix B , as specified by $uplo$.

The dimension of bp must be at least $\max(1, n*(n+1)/2)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

ldz INTEGER. The leading dimension of the output array z ; $ldz \geq 1$. If $jobz = 'V'$, $ldz \geq \max(1, n)$.

$lwork$ INTEGER.

The dimension of the array $work$.

Constraints:

If $n \leq 1$, $lwork \geq 1$;

If $jobz = 'N'$ and $n > 1$, $lwork \geq n$;

If $jobz = 'V'$ and $n > 1$, $lwork \geq 2n$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

$rwork$ REAL for `chpgvd`

DOUBLE PRECISION for `zhpgvd`.

Workspace array, its dimension $\max(1, lrwork)$.

$lrwork$ INTEGER.

The dimension of the array $rwork$.

Constraints:

If $n \leq 1$, $lrwork \geq 1$;

If $jobz = 'N'$ and $n > 1$, $lrwork \geq n$;

If $jobz = 'V'$ and $n > 1$, $lrwork \geq 2n^2 + 5n + 1$.

If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

$iwork$ INTEGER.

Workspace array, its dimension $\max(1, liwork)$.

$liwork$ INTEGER.

The dimension of the array $iwork$.

Constraints:

If $n \leq 1$, $liwork \geq 1$;

If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;
 If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n + 3$.
 If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See [Application Notes](#) for details.

Output Parameters

ap	On exit, the contents of ap are overwritten.
bp	On exit, contains the triangular factor U or L from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$, in the same storage format as B .
w	<p>REAL for chpgvd</p> <p>DOUBLE PRECISION for zhpgvd.</p> <p>Array, DIMENSION at least $\max(1, n)$.</p> <p>If $info = 0$, contains the eigenvalues in ascending order.</p>
z	<p>COMPLEX for chpgvd</p> <p>DOUBLE COMPLEX for zhpgvd.</p> <p>Array $z(ldz, *)$.</p> <p>The second dimension of z must be at least $\max(1, n)$.</p> <p>If $jobz = 'V'$, then if $info = 0$, z contains the matrix Z of eigenvectors. The eigenvectors are normalized as follows:</p> <pre> if $itype = 1$ or 2, $Z^H * B * Z = I$; if $itype = 3$, $Z^H * \text{inv}(B) * Z = I$; </pre> <p>If $jobz = 'N'$, then z is not referenced.</p>
$work(1)$	On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.
$rwork(1)$	On exit, if $info = 0$, then $rwork(1)$ returns the required minimal size of $lrwork$.
$iwork(1)$	On exit, if $info = 0$, then $iwork(1)$ returns the required minimal size of $liwork$.
$info$	<p>INTEGER.</p> <p>If $info = 0$, the execution is successful.</p> <p>If $info = -i$, the i-th argument had an illegal value.</p> <p>If $info > 0$, cpptrf/zpptrf and chpevd/zhpevd returned an error code:</p> <p>If $info = i \leq n$, chpevd/zhpevd failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;</p>

If $\text{info} = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hpgvd` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>bp</code>	Holds the array B of size $(n*(n+1)/2)$.
<code>w</code>	Holds the vector with the number of elements n .
<code>z</code>	Holds the matrix Z of size (n, n) .
<code>itype</code>	Must be 1, 2, or 3. The default value is 1.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>jobz</code>	Restored based on the presence of the argument <code>z</code> as follows: <code>jobz</code> = ' <code>V</code> ', if <code>z</code> is present, <code>jobz</code> = ' <code>N</code> ', if <code>z</code> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of lwork (liwork or lrwork) for the first run or set $\text{lwork} = -1$ ($\text{liwork} = -1$, $\text{lrwork} = -1$).

If you choose the first option and set any of admissible lwork (liwork or lrwork) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (work , iwork , rwork) on exit. Use this value ($\text{work}(1)$, $\text{iwork}(1)$, $\text{rwork}(1)$) for subsequent runs.

If you set $\text{lwork} = -1$ ($\text{liwork} = -1$, $\text{lrwork} = -1$), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (work , iwork , rwork). This operation is called a workspace query.

Note that if you set lwork (liwork , lrwork) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?spgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

FORTRAN 77:

```
call sspgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z,
ldz, work, iwork, ifail, info)
```

```
call dspgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z,
ldz, work, iwork, ifail, info)
```

Fortran 95:

```
call spgvx(ap, bp, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

C:

```
lapack_int LAPACKE_<?>spgvx( int matrix_layout, lapack_int itype, char jobz, char
range, char uplo, lapack_int n, <datatype>* ap, <datatype>* bp, <datatype> vl,
<datatype> vu, lapack_int il, lapack_int iu, <datatype> abstol, lapack_int* m,
<datatype>* w, <datatype>* z, lapack_int ldz, lapack_int* ifail );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

itype INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved:

```
if itype = 1, the problem type is A*x = lambda*B*x;
if itype = 2, the problem type is A*B*x = lambda*x;
if itype = 3, the problem type is B*A*x = lambda*x.
```

jobz CHARACTER*1. Must be 'N' or 'V'.

If *jobz* = 'N', then compute eigenvalues only.

If *jobz* = 'V', then compute eigenvalues and eigenvectors.

range CHARACTER*1. Must be 'A' or 'V' or 'I'.

If *range* = 'A', the routine computes all eigenvalues.

If *range* = 'V', the routine computes eigenvalues *lambda(i)* in the half-open interval:

vl < *lambda(i)* ≤ *vu*.

If *range* = 'I', the routine computes eigenvalues with indices *il* to *iu*.

uplo

CHARACTER*1. Must be 'U' or 'L'.

If $uplo = 'U'$, arrays ap and bp store the upper triangles of A and B ;
 If $uplo = 'L'$, arrays ap and bp store the lower triangles of A and B .

n

INTEGER. The order of the matrices A and B ($n \geq 0$).

ap, bp, work

REAL for sspgvx

DOUBLE PRECISION for dpgvx.

Arrays:

$ap(*)$ contains the packed upper or lower triangle of the symmetric matrix A , as specified by $uplo$.

The dimension of ap must be at least $\max(1, n*(n+1)/2)$.

$bp(*)$ contains the packed upper or lower triangle of the symmetric matrix B , as specified by $uplo$.

The dimension of bp must be at least $\max(1, n*(n+1)/2)$.

$work(*)$ is a workspace array, DIMENSION at least $\max(1, 8n)$.

vl, vu

REAL for sspgvx

DOUBLE PRECISION for dpgvx.

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If $range = 'A'$ or ' I' , vl and vu are not referenced.

il, iu

INTEGER.

If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$

if $n = 0$.

If $range = 'A'$ or ' V' , il and iu are not referenced.

abstol

REAL for sspgvx

DOUBLE PRECISION for dpgvx.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array z . Constraints:

$ldz \geq 1$; if $jobz = 'V'$, $ldz \geq \max(1, n)$.

iwork

INTEGER.

Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

ap

On exit, the contents of ap are overwritten.

<i>bp</i>	On exit, contains the triangular factor U or L from the Cholesky factorization $B = U^T * U$ or $B = L * L^T$, in the same storage format as B .
<i>m</i>	INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$. If <i>range</i> = 'A', $m = n$, and if <i>range</i> = 'I', $m = iu - il + 1$.
<i>w, z</i>	REAL for sspgvx DOUBLE PRECISION for dspgvx. Arrays: <i>w</i> (*), DIMENSION at least max(1, <i>n</i>). If <i>info</i> = 0, contains the eigenvalues in ascending order. <i>z</i> (<i>ldz</i> , *). The second dimension of <i>z</i> must be at least max(1, <i>n</i>). If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix <i>A</i> corresponding to the selected eigenvalues, with the <i>i</i> -th column of <i>z</i> holding the eigenvector associated with <i>w</i> (<i>i</i>). The eigenvectors are normalized as follows: if <i>itype</i> = 1 or 2, $Z^T * B * Z = I$; if <i>itype</i> = 3, $Z^T * \text{inv}(B) * Z = I$; If <i>jobz</i> = 'N', then <i>z</i> is not referenced. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . Note: you must ensure that at least max(1, <i>m</i>) columns are supplied in the array <i>z</i> ; if <i>range</i> = 'V', the exact value of <i>m</i> is not known in advance and an upper bound must be used.
<i>ifail</i>	INTEGER. Array, DIMENSION at least max(1, <i>n</i>). If <i>jobz</i> = 'V', then if <i>info</i> = 0, the first <i>m</i> elements of <i>ifail</i> are zero; if <i>info</i> > 0, the <i>ifail</i> contains the indices of the eigenvectors that failed to converge. If <i>jobz</i> = 'N', then <i>ifail</i> is not referenced.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = <i>-i</i> , the <i>i</i> -th argument had an illegal value. If <i>info</i> > 0, spptrf/dpptrf and sspevx/dspevx returned an error code: If <i>info</i> = <i>i</i> $\leq n$, sspevx/dspevx failed to converge, and <i>i</i> eigenvectors failed to converge. Their indices are stored in the array <i>ifail</i> ; If <i>info</i> = <i>n</i> + <i>i</i> , for $1 \leq i \leq n$, then the leading minor of order <i>i</i> of <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `spgvx` interface are the following:

<code>ap</code>	Holds the array A of size $(n*(n+1)/2)$.
<code>bp</code>	Holds the array B of size $(n*(n+1)/2)$.
<code>w</code>	Holds the vector with the number of elements n .
<code>z</code>	Holds the matrix Z of size (n, n) , where the values n and m are significant.
<code>ifail</code>	Holds the vector with the number of elements n .
<code>itype</code>	Must be 1, 2, or 3. The default value is 1.
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>vl</code>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<code>vu</code>	Default value for this element is $vu = \text{HUGE}(vl)$.
<code>il</code>	Default value for this argument is $il = 1$.
<code>iu</code>	Default value for this argument is $iu = n$.
<code>abstol</code>	Default value for this element is $abstol = 0.0_{\text{WP}}$.
<code>jobz</code>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted. Note that there will be an error condition if <code>ifail</code> is present and z is omitted.
<code>range</code>	Restored based on the presence of arguments vl , vu , il , iu as follows: $range = 'V'$, if one of or both vl and vu are present, $range = 'I'$, if one of or both il and iu are present, $range = 'A'$, if none of vl , vu , il , iu is present. Note that there will be an error condition if one of or both vl and vu are present and at the same time one of or both il and iu are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $abstol + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If `abstol` is less than or equal to zero, then $\epsilon * \|T\|_1$ is used instead, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues are computed most accurately when `abstol` is set to twice the underflow threshold $2 * ?lamch('S')$, not zero.

If this routine returns with `info > 0`, indicating that some eigenvectors did not converge, set `abstol` to $2 * ?lamch('S')$.

?hpgvx

Computes selected eigenvalues and, optionally, eigenvectors of a generalized Hermitian positive-definite eigenproblem with matrices in packed storage.

Syntax**FORTRAN 77:**

```
call chpgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z,
ldz, work, rwork, iwork, ifail, info)

call zhpgvx(itype, jobz, range, uplo, n, ap, bp, vl, vu, il, iu, abstol, m, w, z,
ldz, work, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hpgvx(ap, bp, w [,itype] [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail]
[,abstol] [,info])
```

C:

```
lapack_int LAPACKE_chpgvx( int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, lapack_complex_float* ap, lapack_complex_float* bp, float vl,
float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w,
lapack_complex_float* z, lapack_int ldz, lapack_int* ifail );

lapack_int LAPACKE_zhpgvx( int matrix_layout, lapack_int itype, char jobz, char range,
char uplo, lapack_int n, lapack_complex_double* ap, lapack_complex_double* bp, double vl,
double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w,
lapack_complex_double* z, lapack_int ldz, lapack_int* ifail );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$A^*x = \lambda^*B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x.$$

Here A and B are assumed to be Hermitian, stored in packed format, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

itype	INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: if $itype = 1$, the problem type is $A^*x = \lambda^*B^*x$; if $itype = 2$, the problem type is $A^*B^*x = \lambda^*x$; if $itype = 3$, the problem type is $B^*A^*x = \lambda^*x$.
-------	---

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues <i>lambda</i> (<i>i</i>) in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ap</i> and <i>bp</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>ap</i> and <i>bp</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> (<i>n</i> \geq 0).
<i>ap</i> , <i>bp</i> , <i>work</i>	COMPLEX for chpgvx DOUBLE COMPLEX for zhpgvx. Arrays: <i>ap</i> (*) contains the packed upper or lower triangle of the Hermitian matrix <i>A</i> , as specified by <i>uplo</i> . The dimension of <i>ap</i> must be at least $\max(1, n*(n+1)/2)$. <i>bp</i> (*) contains the packed upper or lower triangle of the Hermitian matrix <i>B</i> , as specified by <i>uplo</i> . The dimension of <i>bp</i> must be at least $\max(1, n*(n+1)/2)$. <i>work</i> (*) is a workspace array, DIMENSION at least $\max(1, 2n)$.
<i>vl</i> , <i>vu</i>	REAL for chpgvx DOUBLE PRECISION for zhpgvx. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il</i> , <i>iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if <i>n</i> > 0; <i>il</i> =1 and <i>iu</i> =0 if <i>n</i> = 0. If <i>range</i> = 'A' or 'V', <i>il</i> and <i>iu</i> are not referenced.
<i>abstol</i>	REAL for chpgvx

DOUBLE PRECISION for zhpgvx.

The absolute error tolerance for the eigenvalues.

See *Application Notes* for more information.

ldz INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', $ldz \geq \max(1, n)$.

rwork REAL for chpgvx

DOUBLE PRECISION for zhpgvx.

Workspace array, DIMENSION at least $\max(1, 7n)$.

iwork INTEGER.

Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

ap On exit, the contents of *ap* are overwritten.

bp On exit, contains the triangular factor *U* or *L* from the Cholesky factorization $B = U^H * U$ or $B = L * L^H$, in the same storage format as *B*.

m INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$. If *range* = 'A', $m = n$, and if *range* = 'I',
 $m = iu - il + 1$.

w REAL for chpgvx

DOUBLE PRECISION for zhpgvx.

Array, DIMENSION at least $\max(1, n)$.

If *info* = 0, contains the eigenvalues in ascending order.

z COMPLEX for chpgvx

DOUBLE COMPLEX for zhpgvx.

Array $z(ldz, *)$.

The second dimension of *z* must be at least $\max(1, n)$.

If *jobz* = 'V', then if *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *A* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w(i)*. The eigenvectors are normalized as follows:

if *itype* = 1 or 2, $Z^H * B * Z = I$;

if *itype* = 3, $Z^H * \text{inv}(B) * Z = I$;

If *jobz* = 'N', then *z* is not referenced.

If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

Note: you must ensure that at least $\max(1,m)$ columns are supplied in the array z ; if $\text{range} = \text{'V'}$, the exact value of m is not known in advance and an upper bound must be used.

ifail

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $\text{jobz} = \text{'V'}$, then if $\text{info} = 0$, the first m elements of *ifail* are zero; if $\text{info} > 0$, the *ifail* contains the indices of the eigenvectors that failed to converge.

If $\text{jobz} = \text{'N'}$, then *ifail* is not referenced.

info

INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the i -th argument had an illegal value.

If $\text{info} > 0$, *cpptrf/zpptrf* and *chpevx/zhpevx* returned an error code:

If $\text{info} = i \leq n$, *chpevx/zhpevx* failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array *ifail*;

If $\text{info} = n + i$, for $1 \leq i \leq n$, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *hpgvx* interface are the following:

*ap*Holds the array A of size $(n*(n+1)/2)$.*bp*Holds the array B of size $(n*(n+1)/2)$.*w*Holds the vector with the number of elements n .*z*Holds the matrix Z of size (n, n) , where the values n and m are significant.*ifail*Holds the vector with the number of elements n .*itype*

Must be 1, 2, or 3. The default value is 1.

*uplo*Must be '*U*' or '*L*'. The default value is '*U*'.*vl*Default value for this element is $vl = -\text{HUGE}(vl)$.*vu*Default value for this element is $vu = \text{HUGE}(vl)$.*il*Default value for this argument is $il = 1$.*iu*Default value for this argument is $iu = n$.*abstol*Default value for this element is $abstol = 0.0_{\text{WP}}$.

jobz Restored based on the presence of the argument *z* as follows:

jobz = 'V', if *z* is present,
jobz = 'N', if *z* is omitted.

Note that there will be an error condition if *ifail* is present and *z* is omitted.

range Restored based on the presence of arguments *vl*, *vu*, *il*, *iu* as follows:

range = 'V', if one of or both *vl* and *vu* are present,
range = 'I', if one of or both *il* and *iu* are present,
range = 'A', if none of *vl*, *vu*, *il*, *iu* is present,

Note that there will be an error condition if one of or both *vl* and *vu* are present and at the same time one of or both *il* and *iu* are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $\text{abstol} + \varepsilon * \max(|a|, |b|)$, where ε is the machine precision.

If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ is used as tolerance, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * ?\text{lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * ?\text{lamch}('S')$.

?sbgv

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

FORTRAN 77:

```
call ssbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, info)
call dsbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, info)
```

Fortran 95:

```
call sbgv(ab, bb, w [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_<?>sbgv( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, <datatype>* ab, lapack_int ldab, <datatype>* bb,
lapack_int ldbb, <datatype>* w, <datatype>* z, lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda * B^*x$. Here A and B are assumed to be symmetric and banded, and B is also positive definite.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in B ($kb \geq 0$).
<i>ab</i> , <i>bb</i> , <i>work</i>	REAL for ssbgv DOUBLE PRECISION for dsbgv Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldb</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix B (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, dimension at least $\max(1, 3n)$
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>ldb</i>	INTEGER. The leading dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; <i>ldz</i> ≥ 1 . If <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$.

Output Parameters

<i>ab</i>	On exit, the contents of <i>ab</i> are overwritten.
<i>bb</i>	On exit, contains the factor S from the split Cholesky factorization $B = S^T * S$, as returned by pbstf/pbstf .
<i>w</i> , <i>z</i>	REAL for ssbgv

DOUBLE PRECISION for `dsbgv`

Arrays:

`w(*)`, DIMENSION at least $\max(1, n)$.

If `info = 0`, contains the eigenvalues in ascending order.

`z(ldz,*)`.

The second dimension of `z` must be at least $\max(1, n)$.

If `jobz = 'V'`, then if `info = 0`, `z` contains the matrix Z of eigenvectors, with the i -th column of `z` holding the eigenvector associated with `w(i)`. The eigenvectors are normalized so that $Z^T * B * Z = I$.

If `jobz = 'N'`, then `z` is not referenced.

`info`

INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the i -th argument had an illegal value.

If `info > 0`, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if `info = n + i`, for $1 \leq i \leq n$, then `pbstf/pbstf` returned `info = i` and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbgv` interface are the following:

`ab` Holds the array A of size $(ka+1, n)$.

`bb` Holds the array B of size $(kb+1, n)$.

`w` Holds the vector with the number of elements n .

`z` Holds the matrix Z of size (n, n) .

`uplo` Must be '`U`' or '`L`'. The default value is '`U`'.

`jobz` Restored based on the presence of the argument `z` as follows:

`jobz = 'V'`, if `z` is present,

`jobz = 'N'`, if `z` is omitted.

?hbgv

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices.

Syntax

FORTRAN 77:

```
call chbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, rwork, info)
call zhbgv(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, rwork, info)
```

Fortran 95:

```
call hbgv(ab, bb, w [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_chbgv( int matrix_layout, char jobz, char uplo, lapack_int n,
                           lapack_int ka, lapack_int kb, lapack_complex_float* ab, lapack_int ldab,
                           lapack_complex_float* bb, lapack_int ldbb, float* w, lapack_complex_float* z,
                           lapack_int ldz );

lapack_int LAPACKE_zhbgv( int matrix_layout, char jobz, char uplo, lapack_int n,
                           lapack_int ka, lapack_int kb, lapack_complex_double* ab, lapack_int ldab,
                           lapack_complex_double* bb, lapack_int ldbb, double* w, lapack_complex_double* z,
                           lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A^*x = \lambda B^*x$. Here A and B are Hermitian and banded matrices, and matrix B is also positive definite.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> (<i>n</i> \geq 0).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> (<i>ka</i> \geq 0).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in <i>B</i> (<i>kb</i> \geq 0).
<i>ab</i> , <i>bb</i> , <i>work</i>	COMPLEX for chbgv

DOUBLE COMPLEX **for** zhbgv

Arrays:

ab (*ldab*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.

The second dimension of the array *ab* must be at least $\max(1, n)$.

bb(*ldb*,*) is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format.

The second dimension of the array *bb* must be at least $\max(1, n)$.

work(*) is a workspace array, dimension at least $\max(1, n)$.

ldab INTEGER. The leading dimension of the array *ab*; must be at least *ka*+1.

*ldb*b INTEGER. The leading dimension of the array *bb*; must be at least *kb*+1.

ldz INTEGER. The leading dimension of the output array *z*; *ldz* ≥ 1 . If *jobz* = 'V', *ldz* $\geq \max(1, n)$.

rwork REAL **for** chbgv

DOUBLE PRECISION **for** zhbgv.

Workspace array, DIMENSION at least $\max(1, 3n)$.

Output Parameters

ab On exit, the contents of *ab* are overwritten.

bb On exit, contains the factor *S* from the split Cholesky factorization *B* = $S^H * S$, as returned by [pbstf/pbstf](#).

w REAL **for** chbgv

DOUBLE PRECISION **for** zhbgv.

Array, DIMENSION at least $\max(1, n)$.

If *info* = 0, contains the eigenvalues in ascending order.

z COMPLEX **for** chbgv

DOUBLE COMPLEX **for** zhbgv

Array *z*(*ldz*,*).

The second dimension of *z* must be at least $\max(1, n)$.

If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w(i)*. The eigenvectors are normalized so that $Z^H * B^* * Z = I$.

If *jobz* = 'N', then *z* is not referenced.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = *-i*, the *i*-th argument had an illegal value.

If *info* > 0, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;
 if $info = n + i$, for $1 \leq i \leq n$, then `pbstf/pbstf` returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbgy` interface are the following:

<code>ab</code>	Holds the array A of size $(ka+1, n)$.
<code>bb</code>	Holds the array B of size $(kb+1, n)$.
<code>w</code>	Holds the vector with the number of elements n .
<code>z</code>	Holds the matrix Z of size (n, n) .
<code>uplo</code>	Must be ' <code>U</code> ' or ' <code>L</code> '. The default value is ' <code>U</code> '.
<code>jobz</code>	Restored based on the presence of the argument <code>z</code> as follows: $jobz = 'V'$, if <code>z</code> is present, $jobz = 'N'$, if <code>z</code> is omitted.

?sbgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

FORTRAN 77:

```
call ssbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, iwork, liwork, info)
call dsbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, iwork, liwork, info)
```

Fortran 95:

```
call sbgvd(ab, bb, w [,uplo] [,z] [,info])
```

C:

```
lapack_int LAPACKE_<?>sbgvd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, <datatype>* ab, lapack_int ldab, <datatype>* bb,
lapack_int ldbb, <datatype>* w, <datatype>* z, lapack_int ldz );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`

- C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda * B^*x$. Here A and B are assumed to be symmetric and banded, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of A and B ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of A and B .
<i>n</i>	INTEGER. The order of the matrices A and B ($n \geq 0$).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in B ($kb \geq 0$).
<i>ab</i> , <i>bb</i> , <i>work</i>	REAL for ssbgvd DOUBLE PRECISION for dsbgvd Arrays: <i>ab</i> (<i>ldab,*</i>) is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldbb,*</i>) is an array containing either upper or lower triangular part of the symmetric matrix B (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i> is a workspace array, its dimension $\max(1, lwork)$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; must be at least $ka+1$.
<i>ldbb</i>	INTEGER. The leading dimension of the array <i>bb</i> ; must be at least $kb+1$.
<i>ldz</i>	INTEGER. The leading dimension of the output array <i>z</i> ; <i>ldz</i> ≥ 1 . If <i>jobz</i> = 'V', <i>ldz</i> $\geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> .

Constraints:

If $n \leq 1$, $lwork \geq 1$;
If $jobz = 'N'$ and $n > 1$, $lwork \geq 3n$;
If $jobz = 'V'$ and $n > 1$, $lwork \geq 2n^2 + 5n + 1$.
If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

$iwork$

INTEGER.

Workspace array, its dimension $\max(1, liwork)$.

$liwork$

INTEGER.

The dimension of the array $iwork$.

Constraints:

If $n \leq 1$, $liwork \geq 1$;
If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;
If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n + 3$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ and $iwork$ arrays, returns these values as the first entries of the $work$ and $iwork$ arrays, and no error message related to $lwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

ab

On exit, the contents of ab are overwritten.

bb

On exit, contains the factor S from the split Cholesky factorization $B = S^T * S$, as returned by [pbstf/pbstf](#).

w, z

REAL for ssbgvd

DOUBLE PRECISION for dsbgvd

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues in ascending order.

$z(ldz, *)$.

The second dimension of z must be at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, z contains the matrix Z of eigenvectors, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^T * B * Z = I$.

If $jobz = 'N'$, then z is not referenced.

<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>iwork(1)</i>	On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value. If <i>info</i> > 0, and if <i>i</i> ≤ <i>n</i> , the algorithm failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero; if <i>info</i> = <i>n</i> + <i>i</i> , for 1 ≤ <i>i</i> ≤ <i>n</i> , then pbstf/pbstf returned <i>info</i> = <i>i</i> and <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `sbgvd` interface are the following:

<i>ab</i>	Holds the array <i>A</i> of size (<i>ka</i> +1, <i>n</i>).
<i>bb</i>	Holds the array <i>B</i> of size (<i>kb</i> +1, <i>n</i>).
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

Application Notes

If it is not clear how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If *lwork* (or *liwork*) has any of admissible sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work(1)*, *iwork(1)*) for subsequent runs.

If *lwork* = -1 (*liwork* = -1), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if *work* (*liwork*) is less than the minimal required value and is not equal to -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?hbгvд

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

FORTRAN 77:

```
call chbgvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, rwork,
lrwork, iwork, liwork, info)
```

```
call zhbqvd(jobz, uplo, n, ka, kb, ab, ldab, bb, ldbb, w, z, ldz, work, lwork, rwork,  
lrwork, iwork, liwork, info)
```

Fortran 95:

```
call hbgvd(ab, bb, w [, uplo] [, z] [, info])
```

C:

```
lapack_int LAPACKE_chbgvd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, lapack_complex_float* ab, lapack_int ldab,
lapack_complex_float* bb, lapack_int ldbb, float* w, lapack_complex_float* z,
lapack_int ldz );
```

```
lapack_int LAPACKE_zhbzvd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, lapack_complex_double* ab, lapack_int ldab,
lapack_complex_double* bb, lapack_int ldbb, double* w, lapack_complex_double* z,
lapack_int ldz );
```

Include Files

- Fortran: mkl.fi
 - Fortran 95: lapack.f90
 - C: mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A^*x = \lambda^*B^*x$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

jobz CHARACTER*1. Must be 'N' or 'V'.

If `jobz = 'N'`, then compute eigenvalues only.

If $jobz = 'v'$, then compute eigenvalues and e

If `uplo = 'U'`, arrays `ab` and `bb` start at `0`.

If $uplo = 'L'$, arrays ab and bb store the lower triangles of A and B .

n INTEGER. The order of the matrices A and B ($n \geq 0$).

ka INTEGER. The number of super- or sub-diagonals in A ($ka \geq 0$).

kb INTEGER. The number of super- or sub-diagonals in B ($kb \geq 0$).

$ab, bb, work$ COMPLEX for chbgvd
DOUBLE COMPLEX for zhbgvd

Arrays:

ab ($ldab,*$) is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by $uplo$) in band storage format.

The second dimension of the array ab must be at least $\max(1, n)$.

bb ($ldbb,*$) is an array containing either upper or lower triangular part of the Hermitian matrix B (as specified by $uplo$) in band storage format.

The second dimension of the array bb must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

$ldab$ INTEGER. The leading dimension of the array ab ; must be at least $ka+1$.

$ldbb$ INTEGER. The leading dimension of the array bb ; must be at least $kb+1$.

ldz INTEGER. The leading dimension of the output array z ; $ldz \geq 1$. If $jobz = 'V'$, $ldz \geq \max(1, n)$.

$lwork$ INTEGER.

The dimension of the array $work$.

Constraints:

If $n \leq 1$, $lwork \geq 1$;

If $jobz = 'N'$ and $n > 1$, $lwork \geq n$;

If $jobz = 'V'$ and $n > 1$, $lwork \geq 2n^2$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

$rwork$ REAL for chbgvd

DOUBLE PRECISION for zhbgvd.

Workspace array, DIMENSION $\max(1, lrwork)$.

$lrwork$ INTEGER.

The dimension of the array $rwork$.

Constraints:

If $n \leq 1$, $lrwork \geq 1$;
 If $jobz = 'N'$ and $n > 1$, $lrwork \geq n$;
 If $jobz = 'V'$ and $n > 1$, $lrwork \geq 2n^2 + 5n + 1$.
 If $lrwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

$iwork$ INTEGER.

Workspace array, DIMENSION $\max(1, liwork)$.

$liwork$ INTEGER.

The dimension of the array $iwork$.

Constraints:

If $n \leq 1$, $lwork \geq 1$;
 If $jobz = 'N'$ and $n > 1$, $liwork \geq 1$;
 If $jobz = 'V'$ and $n > 1$, $liwork \geq 5n + 3$.
 If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$, $rwork$ and $iwork$ arrays, returns these values as the first entries of the $work$, $rwork$ and $iwork$ arrays, and no error message related to $lwork$ or $lrwork$ or $liwork$ is issued by [xerbla](#). See *Application Notes* for details.

Output Parameters

ab On exit, the contents of ab are overwritten.

bb On exit, contains the factor S from the split Cholesky factorization $B = S^H * S$, as returned by [pbstf/pbstf](#).

w REAL for chbgvd
 DOUBLE PRECISION for zhbgvd.

Array, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues in ascending order.

z COMPLEX for chbgvd
 DOUBLE COMPLEX for zhbgvd
 Array $z(Idz,*)$.

The second dimension of z must be at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, z contains the matrix Z of eigenvectors, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^H * B * Z = I$.

If $jobz = 'N'$, then z is not referenced.

<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>rwork(1)</i>	On exit, if <i>info</i> = 0, then <i>rwork(1)</i> returns the required minimal size of <i>lrwork</i> .
<i>iwork(1)</i>	On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value. If <i>info</i> > 0, and if <i>i</i> ≤ <i>n</i> , the algorithm failed to converge, and <i>i</i> off-diagonal elements of an intermediate tridiagonal did not converge to zero; if <i>info</i> = <i>n</i> + <i>i</i> , for 1 ≤ <i>i</i> ≤ <i>n</i> , then <i>pbstf/pbstf</i> returned <i>info</i> = <i>i</i> and <i>B</i> is not positive-definite. The factorization of <i>B</i> could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine *hbqvd* interface are the following:

<i>ab</i>	Holds the array <i>A</i> of size (<i>ka</i> +1, <i>n</i>).
<i>bb</i>	Holds the array <i>B</i> of size (<i>kb</i> +1, <i>n</i>).
<i>w</i>	Holds the vector with the number of elements <i>n</i> .
<i>z</i>	Holds the matrix <i>Z</i> of size (<i>n</i> , <i>n</i>).
<i>uplo</i>	Must be 'U' or 'L'. The default value is 'U'.
<i>jobz</i>	Restored based on the presence of the argument <i>z</i> as follows: <i>jobz</i> = 'V', if <i>z</i> is present, <i>jobz</i> = 'N', if <i>z</i> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (*liwork* or *lrwork*) for the first run or set *lwork* = -1 (*liwork* = -1, *lrwork* = -1).

If you choose the first option and set any of admissible *lwork* (*liwork* or *lrwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*, *rwork*) on exit. Use this value (*work(1)*, *iwork(1)*, *rwork(1)*) for subsequent runs.

If you set `lwork = -1` (`liwork = -1`, `lrwork = -1`), the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (`work`, `iwork`, `rwork`). This operation is called a workspace query.

Note that if you set `lwork` (`liwork`, `lrwork`) to less than the minimal required value and not `-1`, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

?sbgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax

FORTRAN 77:

```
call ssbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu, il, iu,
abstol, m, w, z, ldz, work, iwork, ifail, info)

call dsbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu, il, iu,
abstol, m, w, z, ldz, work, iwork, ifail, info)
```

Fortran 95:

```
call sbgsvx(ab, bb, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q] [,abstol]
[,info])
```

C:

```
lapack_int LAPACKE_<?>sbgsvx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int ka, lapack_int kb, <datatype>* ab, lapack_int ldab,
<datatype>* bb, lapack_int ldbb, <datatype>* q, lapack_int ldq, <datatype> vl,
<datatype> vu, lapack_int il, lapack_int iu, <datatype> abstol, lapack_int* m,
<datatype>* w, <datatype>* z, lapack_int ldz, lapack_int* ifail );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 95: `lapack.f90`
- C: `mkl.h`

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda B^*x$. Here A and B are assumed to be symmetric and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<code>jobz</code>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>V</code> '.
	If <code>jobz</code> = ' <code>N</code> ', then compute eigenvalues only.
	If <code>jobz</code> = ' <code>V</code> ', then compute eigenvalues and eigenvectors.

<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues <i>lambda</i> (<i>i</i>) in the half-open interval: $vl < \lambda(i) \leq vu$. If <i>range</i> = 'I', the routine computes eigenvalues in range <i>il</i> to <i>iu</i> .
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> (<i>n</i> \geq 0).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> (<i>ka</i> \geq 0).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in <i>B</i> (<i>kb</i> \geq 0).
<i>ab</i> , <i>bb</i> , <i>work</i>	REAL for ssbgvx DOUBLE PRECISION for dsbgvx Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldb</i> ,*) is an array containing either upper or lower triangular part of the symmetric matrix <i>B</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, DIMENSION (7*n).
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; must be at least <i>ka</i> +1.
<i>ldb</i>	INTEGER. The leading dimension of the array <i>bb</i> ; must be at least <i>kb</i> +1.
<i>vl</i> , <i>vu</i>	REAL for ssbgvx DOUBLE PRECISION for dsbgvx. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.
<i>il</i> , <i>iu</i>	INTEGER. If <i>range</i> = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $1 \leq il \leq iu \leq n$, if <i>n</i> > 0; <i>il</i> =1 and <i>iu</i> =0 if <i>n</i> = 0.

If $\text{range} = \text{'A'}$ or 'V' , il and iu are not referenced.

$abstol$

REAL for ssbgvx

DOUBLE PRECISION for dsbgvx.

The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz

INTEGER. The leading dimension of the output array z ; $ldz \geq 1$. If $\text{jobz} = \text{'V'}$, $ldz \geq \max(1, n)$.

ldq

INTEGER. The leading dimension of the output array q ; $ldq < 1$.

If $\text{jobz} = \text{'V'}$, $ldq < \max(1, n)$.

$iwork$

INTEGER.

Workspace array, DIMENSION ($5*n$).

Output Parameters

ab

On exit, the contents of ab are overwritten.

bb

On exit, contains the factor S from the split Cholesky factorization $B = S^T * S$, as returned by [pbstf/pbstf](#).

m

INTEGER. The total number of eigenvalues found,

$0 \leq m \leq n$. If $\text{range} = \text{'A'}$, $m = n$, and if $\text{range} = \text{'I'}$,
 $m = iu - il + 1$.

w, z, q

REAL for ssbgvx

DOUBLE PRECISION for dsbgvx

Arrays:

$w(*)$, DIMENSION at least $\max(1, n)$.

If $\text{info} = 0$, contains the eigenvalues in ascending order.

$z(ldz,*)$.

The second dimension of z must be at least $\max(1, n)$.

If $\text{jobz} = \text{'V'}$, then if $\text{info} = 0$, z contains the matrix Z of eigenvectors, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^T * B * Z = I$.

If $\text{jobz} = \text{'N'}$, then z is not referenced.

$q(ldq,*)$.

The second dimension of q must be at least $\max(1, n)$.

If $\text{jobz} = \text{'V'}$, then q contains the n -by- n matrix used in the reduction of $A * x = \lambda * B * x$ to standard form, that is, $C * x = \lambda * x$ and consequently C to tridiagonal form.

If $\text{jobz} = \text{'N'}$, then q is not referenced.

$ifail$

INTEGER.

Array, DIMENSION (m).

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument had an illegal value.

If $info > 0$, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if $info = n + i$, for $1 \leq i \leq n$, then **pbstf/pbstf** returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine **sbgvx** interface are the following:

ab

Holds the array A of size $(ka+1, n)$.

bb

Holds the array B of size $(kb+1, n)$.

w

Holds the vector with the number of elements n .

z

Holds the matrix Z of size (n, n) .

$ifail$

Holds the vector with the number of elements n .

q

Holds the matrix Q of size (n, n) .

$uplo$

Must be ' U ' or ' L '. The default value is ' U '.

vl

Default value for this element is $vl = -\text{HUGE}(v)$.

vu

Default value for this element is $vu = \text{HUGE}(v)$.

il

Default value for this argument is $il = 1$.

iu

Default value for this argument is $iu = n$.

$abstol$

Default value for this element is $abstol = 0.0_{\text{WP}}$.

$jobz$

Restored based on the presence of the argument z as follows:

$jobz = 'V'$, if z is present,

$jobz = 'N'$, if z is omitted.

Note that there will be an error condition if $ifail$ or q is present and z is omitted.

<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: <i>range</i> = 'V', if one or both <i>vl</i> and <i>vu</i> are present, <i>range</i> = 'I', if one of or both <i>il</i> and <i>iu</i> are present, <i>range</i> = 'A', if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present,
Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.	

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $\text{abstol} + \epsilon * \max(|a|, |b|)$, where ϵ is the machine precision.

If *abstol* is less than or equal to zero, then $\epsilon * ||T||_1$ is used as tolerance, where *T* is the tridiagonal matrix obtained by reducing *A* to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * ?\text{lamch}('S')$, not zero.

If this routine returns with *info* > 0, indicating that some eigenvectors did not converge, try setting *abstol* to $2 * ?\text{lamch}('S')$.

?hbgyx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices.

Syntax

FORTRAN 77:

```
call chbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu, il, iu,
abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
call zhbgvx(jobz, range, uplo, n, ka, kb, ab, ldab, bb, ldbb, q, ldq, vl, vu, il, iu,
abstol, m, w, z, ldz, work, rwork, iwork, ifail, info)
```

Fortran 95:

```
call hbgyx(ab, bb, w [,uplo] [,z] [,vl] [,vu] [,il] [,iu] [,m] [,ifail] [,q] [,abstol]
[,info])
```

C:

```
lapack_int LAPACKE_chbgvx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int ka, lapack_int kb, lapack_complex_float* ab, lapack_int ldab,
lapack_complex_float* bb, lapack_int ldbb, lapack_complex_float* q, lapack_int ldq,
float vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float*
w, lapack_complex_float* z, lapack_int ldz, lapack_int* ifail );
lapack_int LAPACKE_zhbgvx( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int ka, lapack_int kb, lapack_complex_double* ab, lapack_int
ldab, lapack_complex_double* bb, lapack_int ldbb, lapack_complex_double* q, lapack_int
ldq, double vl, double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m,
double* w, lapack_complex_double* z, lapack_int ldz, lapack_int* ifail );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90

- C: mkl.h

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A^*x = \lambda B^*x$. Here A and B are assumed to be Hermitian and banded, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobz</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobz</i> = 'N', then compute eigenvalues only. If <i>jobz</i> = 'V', then compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues <i>lambda</i> (<i>i</i>) in the half-open interval: <i>vl</i> < <i>lambda</i> (<i>i</i>) ≤ <i i="" vu<="">. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i>.</i>
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>ab</i> and <i>bb</i> store the upper triangles of <i>A</i> and <i>B</i> ; If <i>uplo</i> = 'L', arrays <i>ab</i> and <i>bb</i> store the lower triangles of <i>A</i> and <i>B</i> .
<i>n</i>	INTEGER. The order of the matrices <i>A</i> and <i>B</i> (<i>n</i> ≥ 0).
<i>ka</i>	INTEGER. The number of super- or sub-diagonals in <i>A</i> (<i>ka</i> ≥ 0).
<i>kb</i>	INTEGER. The number of super- or sub-diagonals in <i>B</i> (<i>kb</i> ≥ 0).
<i>ab</i> , <i>bb</i> , <i>work</i>	COMPLEX for chbgvx DOUBLE COMPLEX for zhbgvx Arrays: <i>ab</i> (<i>ldab</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>A</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>ab</i> must be at least $\max(1, n)$. <i>bb</i> (<i>ldb</i> ,*) is an array containing either upper or lower triangular part of the Hermitian matrix <i>B</i> (as specified by <i>uplo</i>) in band storage format. The second dimension of the array <i>bb</i> must be at least $\max(1, n)$. <i>work</i> (*) is a workspace array, DIMENSION at least $\max(1, n)$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; must be at least <i>ka</i> +1.

lubb INTEGER. The leading dimension of the array *bb*; must be at least $kb+1$.

vl, vu REAL for chbgvx
DOUBLE PRECISION for zhbgvx.
If *range* = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: $vl < vu$.
If *range* = 'A' or 'I', *vl* and *vu* are not referenced.

il, iu INTEGER.
If *range* = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; *il*=1 and *iu*=0 if $n = 0$.
If *range* = 'A' or 'V', *il* and *iu* are not referenced.

abstol REAL for chbgvx
DOUBLE PRECISION for zhbgvx.
The absolute error tolerance for the eigenvalues. See *Application Notes* for more information.

ldz INTEGER. The leading dimension of the output array *z*; $ldz \geq 1$. If *jobz* = 'V', $ldz \geq \max(1, n)$.

ldq INTEGER. The leading dimension of the output array *q*; $ldq \geq 1$. If *jobz* = 'V', $ldq \geq \max(1, n)$.

rwork REAL for chbgvx
DOUBLE PRECISION for zhbgvx.
Workspace array, DIMENSION at least $\max(1, 7n)$.

iwork INTEGER.
Workspace array, DIMENSION at least $\max(1, 5n)$.

Output Parameters

ab On exit, the contents of *ab* are overwritten.

bb On exit, contains the factor *S* from the split Cholesky factorization $B = S^H * S$, as returned by **pbstf/pbstf**.

m INTEGER. The total number of eigenvalues found,
 $0 \leq m \leq n$. If *range* = 'A', *m* = *n*, and if *range* = 'I',
 $m = iu - il + 1$.

w REAL for chbgvx
DOUBLE PRECISION for zhbgvx.

Array $w(*)$, DIMENSION at least $\max(1, n)$.

If $info = 0$, contains the eigenvalues in ascending order.

z, q

COMPLEX for chbgvx

DOUBLE COMPLEX for zhbgvx

Arrays:

$z(Idz,*)$.

The second dimension of z must be at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, z contains the matrix Z of eigenvectors, with the i -th column of z holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^H * B^* Z = I$.

If $jobz = 'N'$, then z is not referenced.

$q(Idq,*)$.

The second dimension of q must be at least $\max(1, n)$.

If $jobz = 'V'$, then q contains the n -by- n matrix used in the reduction of $Ax = \lambda Bx$ to standard form, that is, $Cx = \lambda x$ and consequently C to tridiagonal form.

If $jobz = 'N'$, then q is not referenced.

$ifail$

INTEGER.

Array, DIMENSION at least $\max(1, n)$.

If $jobz = 'V'$, then if $info = 0$, the first m elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th argument had an illegal value.

If $info > 0$, and

if $i \leq n$, the algorithm failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if $info = n + i$, for $1 \leq i \leq n$, then pbstf/pbstf returned $info = i$ and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `hbgyx` interface are the following:

ab

Holds the array A of size $(ka+1, n)$.

<i>bb</i>	Holds the array B of size $(kb+1, n)$.
<i>w</i>	Holds the vector with the number of elements n .
<i>z</i>	Holds the matrix Z of size (n, n) .
<i>ifail</i>	Holds the vector with the number of elements n .
<i>q</i>	Holds the matrix Q of size (n, n) .
<i>uplo</i>	Must be ' U ' or ' L '. The default value is ' U '.
<i>vl</i>	Default value for this element is $vl = -\text{HUGE}(vl)$.
<i>vu</i>	Default value for this element is $vu = \text{HUGE}(vl)$.
<i>il</i>	Default value for this argument is $il = 1$.
<i>iu</i>	Default value for this argument is $iu = n$.
<i>abstol</i>	Default value for this element is $abstol = 0.0_{\text{WP}}$.
<i>jobz</i>	Restored based on the presence of the argument z as follows: $jobz = 'V'$, if z is present, $jobz = 'N'$, if z is omitted. Note that there will be an error condition if <i>ifail</i> or <i>q</i> is present and <i>z</i> is omitted.
<i>range</i>	Restored based on the presence of arguments <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> as follows: $range = 'V'$, if one of or both <i>vl</i> and <i>vu</i> are present, $range = 'I'$, if one of or both <i>il</i> and <i>iu</i> are present, $range = 'A'$, if none of <i>vl</i> , <i>vu</i> , <i>il</i> , <i>iu</i> is present. Note that there will be an error condition if one of or both <i>vl</i> and <i>vu</i> are present and at the same time one of or both <i>il</i> and <i>iu</i> are present.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $\text{abstol} + \varepsilon * \max(|a|, |b|)$, where ε is the machine precision.

If *abstol* is less than or equal to zero, then $\varepsilon * \|T\|_1$ will be used in its place, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when *abstol* is set to twice the underflow threshold $2 * ?\text{lamch}('S')$, not zero.

If this routine returns with *info* > 0 , indicating that some eigenvectors did not converge, try setting *abstol* to $2 * ?\text{lamch}('S')$.

Generalized Nonsymmetric Eigenproblems

This section describes LAPACK driver routines used for solving generalized nonsymmetric eigenproblems. See also [computational routines](#) that can be called to solve these problems. [Table "Driver Routines for Solving Generalized Nonsymmetric Eigenproblems"](#) lists all such driver routines for the FORTRAN 77 interface. The corresponding routine names in the Fortran 95 interface are without the first symbol.

Driver Routines for Solving Generalized Nonsymmetric Eigenproblems

Routine Name	Operation performed
gges	Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.
ggesx	Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.
ggev	Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.
ggevx	Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

?gges

Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.

Syntax**FORTRAN 77:**

```
call sgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphar, alphai,
beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info)

call dgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alphar, alphai,
beta, vsl, ldvsl, vsr, ldvsr, work, lwork, bwork, info)

call cgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta, vsl,
ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info)

call zgges(jobvsl, jobvsr, sort, selctg, n, a, lda, b, ldb, sdim, alpha, beta, vsl,
ldvsl, vsr, ldvsr, work, lwork, rwork, bwork, info)
```

Fortran 95:

```
call gges(a, b, alphar, alphai, beta [,vsl] [,vsr] [,select] [,sdim] [,info])
call gges(a, b, alpha, beta [, vsl] [,vsr] [,select] [,sdim] [,info])
```

C:

```
lapack_int LAPACKE_sgges( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_S_SELECT3 select, lapack_int n, float* a, lapack_int lda, float* b, lapack_int
ldb, lapack_int* sdim, float* alphar, float* alphai, float* beta, float* vsl,
lapack_int ldvsl, float* vsr, lapack_int ldvsr );

lapack_int LAPACKE_dgges( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_D_SELECT3 select, lapack_int n, double* a, lapack_int lda, double* b,
lapack_int ldb, lapack_int* sdim, double* alphar, double* alphai, double* beta,
double* vsl, lapack_int ldvsl, double* vsr, lapack_int ldvsr );

lapack_int LAPACKE_cgges( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_C_SELECT2 select, lapack_int n, lapack_complex_float* a, lapack_int lda,
lapack_complex_float* b, lapack_int ldb, lapack_int* sdim, lapack_complex_float* alpha,
lapack_complex_float* beta, lapack_complex_float* vsl, lapack_int ldvsl,
lapack_complex_float* vsr, lapack_int ldvsr );
```

```
lapack_int LAPACKE_zgges( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_Z_SELECT2 select, lapack_int n, lapack_complex_double* a, lapack_int lda,
lapack_complex_double* b, lapack_int ldb, lapack_int* sdim, lapack_complex_double*
alpha, lapack_complex_double* beta, lapack_complex_double* vsl, lapack_int ldvsl,
lapack_complex_double* vsr, lapack_int ldvsr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The ?gges routine computes the generalized eigenvalues, the generalized real/complex Schur form (S, T), optionally, the left and/or right matrices of Schur vectors (vsl and vsr) for a pair of n -by- n real/complex nonsymmetric matrices (A, B). This gives the generalized Schur factorization

$$(A, B) = (vsl^*S * vsr^H, vsl^*T^* vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T . The leading columns of vsl and vsr then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

If only the generalized eigenvalues are needed, use the driver [ggev](#) instead, which is faster.

A generalized eigenvalue for a pair of matrices (A, B) is a scalar w or a ratio $\alpha / \beta = w$, such that $A - w^*B$ is singular. It is usually represented as the pair (α, β), as there is a reasonable interpretation for $\beta=0$ or for both being zero. A pair of matrices (S, T) is in the generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S are "standardized" by making the corresponding elements of T have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (S, T) is in generalized complex Schur form if S and T are upper triangular and, in addition, the diagonal of T are non-negative real numbers.

The ?gges routine replaces the deprecated ?gegs routine.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobvsl</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvsl</i> = 'N', then the left Schur vectors are not computed. If <i>jobvsl</i> = 'V', then the left Schur vectors are computed.
<i>jobvsr</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvsr</i> = 'N', then the right Schur vectors are not computed.

If $\text{jobvsr} = 'V'$, then the right Schur vectors are computed.

sort

CHARACTER*1. Must be ' N ' or ' S '. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.

If $\text{sort} = 'N'$, then eigenvalues are not ordered.

If $\text{sort} = 'S'$, eigenvalues are ordered (see selctg).

selctg

LOGICAL FUNCTION of three REAL arguments for real flavors.

LOGICAL FUNCTION of two COMPLEX arguments for complex flavors.

selctg must be declared EXTERNAL in the calling subroutine.

If $\text{sort} = 'S'$, selctg is used to select eigenvalues to sort to the top left of the Schur form.

If $\text{sort} = 'N'$, selctg is not referenced.

For real flavors:

An eigenvalue $(\text{alphar}(j) + \text{alphai}(j))/\text{beta}(j)$ is selected if $\text{selctg}(\text{alphar}(j), \text{alphai}(j), \text{beta}(j))$ is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy $\text{selctg}(\text{alphar}(j), \text{alphai}(j), \text{beta}(j)) = .TRUE.$ after ordering. In this case info is set to $n+2$.

For complex flavors:

An eigenvalue $\text{alpha}(j) / \text{beta}(j)$ is selected if $\text{selctg}(\text{alpha}(j), \text{beta}(j))$ is true.

Note that a selected complex eigenvalue may no longer satisfy $\text{selctg}(\text{alpha}(j), \text{beta}(j)) = .TRUE.$ after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case info is set to $n+2$ (see info below).

n

INTEGER. The order of the matrices A , B , vsl , and vsr ($n \geq 0$).

a, b, work

REAL for sgges

DOUBLE PRECISION for dgges

COMPLEX for cgges

DOUBLE COMPLEX for zgges.

Arrays:

$a(l\text{da},*)$ is an array containing the n -by- n matrix A (first of the pair of matrices).

The second dimension of a must be at least $\max(1, n)$.

$b(l\text{db},*)$ is an array containing the n -by- n matrix B (second of the pair of matrices).

The second dimension of b must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, l\text{work})$.

$l\text{da}$

INTEGER. The leading dimension of the array a . Must be at least $\max(1, n)$.

<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> . Must be at least $\max(1, n)$.
<i>ldvsl, ldvsr</i>	INTEGER. The leading dimensions of the output matrices <i>vsl</i> and <i>vsr</i> , respectively. Constraints: <i>ldvsl</i> ≥ 1 . If <i>jobvsl</i> = 'V', <i>ldvsl</i> $\geq \max(1, n)$. <i>ldvsr</i> ≥ 1 . If <i>jobvsr</i> = 'V', <i>ldvsr</i> $\geq \max(1, n)$.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . <i>lwork</i> $\geq \max(1, 8n+16)$ for real flavors; <i>lwork</i> $\geq \max(1, 2n)$ for complex flavors. For good performance, <i>lwork</i> must generally be larger. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued by <i>xerbla</i> .
<i>rwork</i>	REAL for <i>cgges</i> DOUBLE PRECISION for <i>zgges</i> Workspace array, DIMENSION at least $\max(1, 8n)$. This array is used in complex flavors only.
<i>bwork</i>	LOGICAL. Workspace array, DIMENSION at least $\max(1, n)$. Not referenced if <i>sort</i> = 'N'.

Output Parameters

<i>a</i>	On exit, this array has been overwritten by its generalized Schur form <i>S</i> .
<i>b</i>	On exit, this array has been overwritten by its generalized Schur form <i>T</i> .
<i>sdim</i>	INTEGER. If <i>sort</i> = 'N', <i>sdim</i> = 0. If <i>sort</i> = 'S', <i>sdim</i> is equal to the number of eigenvalues (after sorting) for which <i>selctg</i> is true. Note that for real flavors complex conjugate pairs for which <i>selctg</i> is true for either eigenvalue count as 2.
<i>alphar, alphai</i>	REAL for <i>sgges</i> ; DOUBLE PRECISION for <i>dgges</i> . Arrays, DIMENSION at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors. See <i>beta</i> .
<i>alpha</i>	COMPLEX for <i>cgges</i> ;

DOUBLE COMPLEX for zgges.

Array, DIMENSION at least max(1, n). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta

REAL for sgges

DOUBLE PRECISION for dgges

COMPLEX for cgges

DOUBLE COMPLEX for zgges.

Array, DIMENSION at least max(1, n).

For real flavors:

On exit, $(\text{alphar}(j) + \text{alphaai}(j)*i)/\text{beta}(j)$, $j=1, \dots, n$, will be the generalized eigenvalues.

*alphar(j) + alphaai(j)*i* and *beta(j)*, $j=1, \dots, n$ are the diagonals of the complex Schur form (*S,T*) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (*A,B*) were further reduced to triangular form using complex unitary transformations. If *alphaai(j)* is zero, then the *j*-th eigenvalue is real; if positive, then the *j*-th and (*j+1*)-st eigenvalues are a complex conjugate pair, with *alphaai(j+1)* negative.

For complex flavors:

On exit, *alpha(j)/beta(j)*, $j=1, \dots, n$, will be the generalized eigenvalues.

alpha(j), $j=1, \dots, n$, and *beta(j)*, $j=1, \dots, n$ are the diagonals of the complex Schur form (*S,T*) output by cgges/zgges. The *beta(j)* will be non-negative real.

See also *Application Notes* below.

vsl, vsr

REAL for sgges

DOUBLE PRECISION for dgges

COMPLEX for cgges

DOUBLE COMPLEX for zgges.

Arrays:

vsl(Idvsl,),* the second dimension of *vs*l must be at least max(1, *n*).

If *jobvsl* = 'V', this array will contain the left Schur vectors.

If *jobvsl* = 'N', *vs*l is not referenced.

vsr(Idvsr,),* the second dimension of *vsr* must be at least max(1, *n*).

If *jobvsr* = 'V', this array will contain the right Schur vectors.

If *jobvsr* = 'N', *vsr* is not referenced.

work(1)

On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*th parameter had an illegal value.

If *info* = *i*, and

$i \leq n$:

the QZ iteration failed. (A, B) is not in Schur form, but $\text{alphar}(j)$, $\text{alphai}(j)$ (for real flavors), or $\text{alpha}(j)$ (for complex flavors), and $\text{beta}(j)$, $j=info+1, \dots, n$ should be correct.

$i > n$: errors that usually indicate LAPACK problems:

$i = n+1$: other than QZ iteration failed in `hgeqz`;

$i = n+2$: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy $\text{selectg} = .TRUE..$ This could also be caused due to scaling;

$i = n+3$: reordering failed in `tgsen`.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `gges` interface are the following:

a	Holds the matrix A of size (n, n) .
b	Holds the matrix B of size (n, n) .
alphar	Holds the vector of length n . Used in real flavors only.
alphai	Holds the vector of length n . Used in real flavors only.
alpha	Holds the vector of length n . Used in complex flavors only.
beta	Holds the vector of length n .
vsl	Holds the matrix VSL of size (n, n) .
vsr	Holds the matrix VSR of size (n, n) .
jobvsl	Restored based on the presence of the argument vsl as follows: $\text{jobvsl} = 'V'$, if vsl is present, $\text{jobvsl} = 'N'$, if vsl is omitted.
jobvsr	Restored based on the presence of the argument vsr as follows: $\text{jobvsr} = 'V'$, if vsr is present, $\text{jobvsr} = 'N'$, if vsr is omitted.
sort	Restored based on the presence of the argument select as follows: $\text{sort} = 'S'$, if select is present, $\text{sort} = 'N'$, if select is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of lwork for the first run or set $\text{lwork} = -1$.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work(1)*) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar(j)/beta(j)* and *alphai(j)/beta(j)* may easily over- or underflow, and *beta(j)* may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* will be always less than and usually comparable with *norm(A)* in magnitude, and *beta* always less than and usually comparable with *norm(B)*.

?ggesx

Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.

Syntax

FORTRAN 77:

```
call sggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alphar,
alphai, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, iwork, liwork,
bwork, info)

call dggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alphar,
alphai, beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, iwork, liwork,
bwork, info)

call cggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alpha,
beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, rwork, iwork, liwork,
bwork, info)

call zggesx (jobvsl, jobvsr, sort, selctg, sense, n, a, lda, b, ldb, sdim, alpha,
beta, vsl, ldvsl, vsr, ldvsr, rconde, rcondv, work, lwork, rwork, iwork, liwork,
bwork, info)
```

Fortran 95:

```
call ggesx(a, b, alphar, alphai, beta [,vsl] [,vsr] [,select] [,sdim] [,rconde] [,rcondv] [,info])

call ggesx(a, b, alpha, beta [, vsl] [,vsr] [,select] [,sdim] [,rconde] [,rcondv] [,info])
```

C:

```
lapack_int LAPACKE_sggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_S_SELECT3 select, char sense, lapack_int n, float* a, lapack_int lda, float* b,
lapack_int ldb, lapack_int* sdim, float* alphar, float* alphai, float* beta, float*
vsl, lapack_int ldvsl, float* vsr, lapack_int ldvsr, float* rconde, float* rcondv );

lapack_int LAPACKE_dggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_D_SELECT3 select, char sense, lapack_int n, double* a, lapack_int lda, double*
b, lapack_int ldb, lapack_int* sdim, double* alphar, double* alphai, double* beta,
double* vsl, lapack_int ldvsl, double* vsr, lapack_int ldvsr, double* rconde, double*
rcondv );
```

```

lapack_int LAPACKE_cggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_C_SELECT2 select, char sense, lapack_int n, lapack_complex_float* a, lapack_int
lda, lapack_complex_float* b, lapack_int ldb, lapack_int* sdim, lapack_complex_float*
alpha, lapack_complex_float* beta, lapack_complex_float* vsl, lapack_int ldvsl,
lapack_complex_float* vsr, lapack_int ldvsr, float* rconde, float* rcondv );

lapack_int LAPACKE_zggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_Z_SELECT2 select, char sense, lapack_int n, lapack_complex_double* a, lapack_int
lda, lapack_complex_double* b, lapack_int ldb, lapack_int* sdim, lapack_complex_double*
alpha, lapack_complex_double* beta, lapack_complex_double* vsl, lapack_int ldvsl,
lapack_complex_double* vsr, lapack_int ldvsr, double* rconde, double* rcondv );

```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes for a pair of n -by- n real/complex nonsymmetric matrices (A, B), the generalized eigenvalues, the generalized real/complex Schur form (S, T), optionally, the left and/or right matrices of Schur vectors (vsl and vsr). This gives the generalized Schur factorization

$$(A, B) = (vsl^* S * vsr^H, vsl^* T^* vsr^H)$$

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix S and the upper triangular matrix T ; computes a reciprocal condition number for the average of the selected eigenvalues ($rconde$); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues ($rcondv$). The leading columns of vsl and vsr then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices (A, B) is a scalar w or a ratio $\alpha / \beta = w$, such that $A - wB$ is singular. It is usually represented as the pair (α, β), as there is a reasonable interpretation for $\beta=0$ or for both being zero. A pair of matrices (S, T) is in generalized real Schur form if T is upper triangular with non-negative diagonal and S is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of S will be "standardized" by making the corresponding elements of T have the form:

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

and the pair of corresponding 2-by-2 blocks in S and T will have a complex conjugate pair of generalized eigenvalues. A pair of matrices (S, T) is in generalized complex Schur form if S and T are upper triangular and, in addition, the diagonal of T are non-negative real numbers.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

jobvsl

CHARACTER*1. Must be 'N' or 'V'.

If *jobvsl* = 'N', then the left Schur vectors are not computed.

If $\text{jobvsl} = \text{'V'}$, then the left Schur vectors are computed.

jobvsr

CHARACTER*1. Must be ' N ' or ' V '.

If $\text{jobvsr} = \text{'N'}$, then the right Schur vectors are not computed.

If $\text{jobvsr} = \text{'V'}$, then the right Schur vectors are computed.

sort

CHARACTER*1. Must be ' N ' or ' S '. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.

If $\text{sort} = \text{'N'}$, then eigenvalues are not ordered.

If $\text{sort} = \text{'S'}$, eigenvalues are ordered (see selctg).

selctg

LOGICAL FUNCTION of three REAL arguments for real flavors.

LOGICAL FUNCTION of two COMPLEX arguments for complex flavors.

selctg must be declared EXTERNAL in the calling subroutine.

If $\text{sort} = \text{'S'}$, selctg is used to select eigenvalues to sort to the top left of the Schur form.

If $\text{sort} = \text{'N'}$, selctg is not referenced.

For real flavors:

An eigenvalue $(\text{alphar}(j) + \text{alphai}(j))/\text{beta}(j)$ is selected if $\text{selctg}(\text{alphar}(j), \text{alphai}(j), \text{beta}(j))$ is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy $\text{selctg}(\text{alpha}(j), \text{beta}(j)) = .\text{TRUE}.$ after ordering. In this case info is set to $n+2$.

For complex flavors:

An eigenvalue $\text{alpha}(j) / \text{beta}(j)$ is selected if $\text{selctg}(\text{alpha}(j), \text{beta}(j))$ is true.

Note that a selected complex eigenvalue may no longer satisfy $\text{selctg}(\text{alpha}(j), \text{beta}(j)) = .\text{TRUE}.$ after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case info is set to $n+2$ (see info below).

sense

CHARACTER*1. Must be ' N ', ' E ', ' V ', or ' B '. Determines which reciprocal condition number are computed.

If $\text{sense} = \text{'N'}$, none are computed;

If $\text{sense} = \text{'E'}$, computed for average of selected eigenvalues only;

If $\text{sense} = \text{'V'}$, computed for selected deflating subspaces only;

If $\text{sense} = \text{'B'}$, computed for both.

If sense is ' E ', ' V ', or ' B ', then sort must equal ' S '.

n

INTEGER. The order of the matrices A , B , vsl , and vsr ($n \geq 0$).

a , b , work

REAL for sggesx

DOUBLE PRECISION for dggesx

COMPLEX for cggesx

DOUBLE COMPLEX for zggesx.

Arrays:

a(*lda*,*) is an array containing the n -by- n matrix *A* (first of the pair of matrices).

The second dimension of *a* must be at least $\max(1, n)$.

b(*ldb*,*) is an array containing the n -by- n matrix *B* (second of the pair of matrices).

The second dimension of *b* must be at least $\max(1, n)$.

work is a workspace array, its dimension $\max(1, \ iwork)$.

lda

INTEGER. The leading dimension of the array *a*.

Must be at least $\max(1, n)$.

ldb

INTEGER. The leading dimension of the array *b*.

Must be at least $\max(1, n)$.

ldvsl, ldvsr

INTEGER. The leading dimensions of the output matrices *vsl* and *vsr*, respectively. Constraints:

ldvsl ≥ 1 . If *jobvsl* = 'V', *ldvsl* $\geq \max(1, n)$.

ldvsr ≥ 1 . If *jobvsr* = 'V', *ldvsr* $\geq \max(1, n)$.

iwork

INTEGER.

The dimension of the array *work*.

For real flavors:

If *n*=0 then *iwork* ≥ 1 .

If *n*>0 and *sense* = 'N', then *iwork* $\geq \max(8*n, 6*n+16)$.

If *n*>0 and *sense* = 'E', 'V', or 'B', then *iwork* $\geq \max(8*n, 6*n+16, 2*sdim*(n-sdim))$;

For complex flavors:

If *n*=0 then *iwork* ≥ 1 .

If *n*>0 and *sense* = 'N', then *iwork* $\geq \max(1, 2*n)$;

If *n*>0 and *sense* = 'E', 'V', or 'B', then *iwork* $\geq \max(1, 2*n, 2*sdim*(n-sdim))$.

Note that $2*sdim*(n-sdim) \leq n*n/2$.

An error is only returned if *iwork* $< \max(8*n, 6*n+16)$ for real flavors, and *iwork* $< \max(1, 2*n)$ for complex flavors, but if *sense* = 'E', 'V', or 'B', this may not be large enough.

If *iwork*=-1, then a workspace query is assumed; the routine only calculates the bound on the optimal size of the *work* array and the minimum size of the *iwork* array, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *iwork* or *liwork* is issued by xerbla.

rwork

REAL for cggesx

DOUBLE PRECISION for zggesx

Workspace array, DIMENSION at least max(1, 8n).

This array is used in complex flavors only.

iwork

INTEGER.

Workspace array, DIMENSION max(1, *liwork*).

liwork

INTEGER.

The dimension of the array *iwork*.

If *sense* = 'N', or *n*=0, then *liwork*≥1,

otherwise *liwork* ≥ (n+6) for real flavors, and *liwork* ≥ (n+2) for complex flavors.

If *liwork*=-1, then a workspace query is assumed; the routine only calculates the bound on the optimal size of the *work* array and the minimum size of the *iwork* array, returns these values as the first entries of the *work* and *iwork* arrays, and no error message related to *lwork* or *liwork* is issued by xerbla.

bwork

LOGICAL.

Workspace array, DIMENSION at least max(1, *n*).

Not referenced if *sort* = 'N'.

Output Parameters

a

On exit, this array has been overwritten by its generalized Schur form *S*.

b

On exit, this array has been overwritten by its generalized Schur form *T*.

sdim

INTEGER.

If *sort* = 'N', *sdim*= 0.

If *sort* = 'S', *sdim* is equal to the number of eigenvalues (after sorting) for which *selectg* is true.

Note that for real flavors complex conjugate pairs for which *selectg* is true for either eigenvalue count as 2.

alphar, *alphai*

REAL for sggesx;

DOUBLE PRECISION for dggesx.

Arrays, DIMENSION at least max(1, *n*) each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

alpha

COMPLEX for cggesx;

DOUBLE COMPLEX for zggesx.

Array, DIMENSION at least max(1, *n*). Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta

REAL for sggesx

DOUBLE PRECISION for dggesx

COMPLEX for cggesx

DOUBLE COMPLEX for zggesx.

Array, DIMENSION at least max(1, n).

For real flavors:

On exit, (*alphar(j)* + *alphai(j)*i*)/*beta(j)*, j=1,..., n will be the generalized eigenvalues.

alphar(j) + *alphai(j)*i* and *beta(j)*, j=1,..., n are the diagonals of the complex Schur form (*S,T*) that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of (*A,B*) were further reduced to triangular form using complex unitary transformations. If *alphai(j)* is zero, then the j-th eigenvalue is real; if positive, then the j-th and (j+1)-st eigenvalues are a complex conjugate pair, with *alphai(j+1)* negative.

For complex flavors:

On exit, *alpha(j)/beta(j)*, j=1,..., n will be the generalized eigenvalues. *alpha(j)*, j=1,..., n, and *beta(j)*, j=1,...,n are the diagonals of the complex Schur form (*S,T*) output by cggesx/zggesx. The *beta(j)* will be non-negative real.

See also *Application Notes* below.

vsl, vsr

REAL for sggesx

DOUBLE PRECISION for dggesx

COMPLEX for cggesx

DOUBLE COMPLEX for zggesx.

Arrays:

vsl(*ldvsl,**), the second dimension of *vsl* must be at least max(1, n).

If *jobvsl* = 'V', this array will contain the left Schur vectors.

If *jobvsl* = 'N', *vsl* is not referenced.

vsr(*ldvsr,**), the second dimension of *vsr* must be at least max(1, n).

If *jobvsr* = 'V', this array will contain the right Schur vectors.

If *jobvsr* = 'N', *vsr* is not referenced.

rconde, rcondv

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION (2) each

If *sense* = 'E' or 'B', *rconde*(1) and *rconde*(2) contain the reciprocal condition numbers for the average of the selected eigenvalues.

Not referenced if *sense* = 'N' or 'V'.

If *sense* = 'V' or 'B', *rcondv*(1) and *rcondv*(2) contain the reciprocal condition numbers for the selected deflating subspaces.

Not referenced if *sense* = 'N' or 'E'.

<i>work(1)</i>	On exit, if <i>info</i> = 0, then <i>work(1)</i> returns the required minimal size of <i>lwork</i> .
<i>iwork(1)</i>	On exit, if <i>info</i> = 0, then <i>iwork(1)</i> returns the required minimal size of <i>liwork</i> .
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> th parameter had an illegal value. If <i>info</i> = <i>i</i> , and <i>i</i> ≤ <i>n</i> : the QZ iteration failed. (<i>A</i> , <i>B</i>) is not in Schur form, but <i>alphar(j)</i> , <i>alphai(j)</i> (for real flavors), or <i>alpha(j)</i> (for complex flavors), and <i>beta(j)</i> , <i>j</i> = <i>info</i> +1, ..., <i>n</i> should be correct. <i>i</i> > <i>n</i> : errors that usually indicate LAPACK problems: <i>i</i> = <i>n</i> +1: other than QZ iteration failed in ?hgeqz; <i>i</i> = <i>n</i> +2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy <i>selctg</i> = .TRUE.. This could also be caused due to scaling; <i>i</i> = <i>n</i> +3: reordering failed in <i>tgsen</i> .

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggesx` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>alphar</i>	Holds the vector of length <i>n</i> . Used in real flavors only.
<i>alphai</i>	Holds the vector of length <i>n</i> . Used in real flavors only.
<i>alpha</i>	Holds the vector of length <i>n</i> . Used in complex flavors only.
<i>beta</i>	Holds the vector of length <i>n</i> .
<i>vsl</i>	Holds the matrix <i>VSL</i> of size (<i>n</i> , <i>n</i>).
<i>vsr</i>	Holds the matrix <i>VSR</i> of size (<i>n</i> , <i>n</i>).
<i>rconde</i>	Holds the vector of length (2).
<i>rcondv</i>	Holds the vector of length (2).
<i>jobvsl</i>	Restored based on the presence of the argument <i>vs</i> / <i>l</i> as follows: <i>jobvsl</i> = 'V', if <i>vs</i> / <i>l</i> is present,

	<i>jobvsl</i> = 'N', if <i>vsl</i> is omitted.
<i>jobvsr</i>	Restored based on the presence of the argument <i>vsr</i> as follows: <i>jobvsr</i> = 'V', if <i>vsr</i> is present, <i>jobvsr</i> = 'N', if <i>vsr</i> is omitted.
<i>sort</i>	Restored based on the presence of the argument <i>select</i> as follows: <i>sort</i> = 'S', if <i>select</i> is present, <i>sort</i> = 'N', if <i>select</i> is omitted.
<i>sense</i>	Restored based on the presence of arguments <i>rconde</i> and <i>rcondv</i> as follows: <i>sense</i> = 'B', if both <i>rconde</i> and <i>rcondv</i> are present, <i>sense</i> = 'E', if <i>rconde</i> is present and <i>rcondv</i> omitted, <i>sense</i> = 'V', if <i>rconde</i> is omitted and <i>rcondv</i> present, <i>sense</i> = 'N', if both <i>rconde</i> and <i>rcondv</i> are omitted.

Note that there will be an error condition if *rconde* or *rcondv* are present and *select* is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* (or *liwork*) for the first run or set *lwork* = -1 (*liwork* = -1).

If you choose the first option and set any of admissible *lwork* (or *liwork*) sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*) on exit. Use this value (*work*(1), *iwork*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*, *iwork*). This operation is called a workspace query.

Note that if you set *lwork* (*liwork*) to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar(j)/beta(j)* and *alphai(j)/beta(j)* may easily over- or underflow, and *beta(j)* may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* will be always less than and usually comparable with *norm(A)* in magnitude, and *beta* always less than and usually comparable with *norm(B)*.

?ggev

Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.

Syntax

FORTRAN 77:

```
call sggev(jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta, vl, ldvl, vr, ldvr,
work, lwork, info)

call dggev(jobvl, jobvr, n, a, lda, b, ldb, alphar, alphai, beta, vl, ldvl, vr, ldvr,
work, lwork, info)

call cggev(jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, work,
lwork, rwork, info)
```

```
call zggev(jobvl, jobvr, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr, ldvr, work,
lwork, rwork, info)
```

Fortran 95:

```
call ggev(a, b, alphar, alphai, beta [,vl] [,vr] [,info])
call ggev(a, b, alpha, beta [, vl] [,vr] [,info])
```

C:

```
lapack_int LAPACKE_sggev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
float* a, lapack_int lda, float* b, lapack_int ldb, float* alphar, float* alphai,
float* beta, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr );
lapack_int LAPACKE_dggev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
double* a, lapack_int lda, double* b, lapack_int ldb, double* alphar, double* alphai,
double* beta, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr );
lapack_int LAPACKE_cggev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* alpha, lapack_complex_float* beta, lapack_complex_float* vl,
lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr );
lapack_int LAPACKE_zggev( int matrix_layout, char jobvl, char jobvr, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* alpha, lapack_complex_double* beta, lapack_complex_double* vl,
lapack_int ldvl, lapack_complex_double* vr, lapack_int ldvr );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The ?ggev routine computes the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors for a pair of n -by- n real/complex nonsymmetric matrices (A,B).

A generalized eigenvalue for a pair of matrices (A,B) is a scalar λ or a ratio $\alpha/\beta = \lambda$, such that $A - \lambda B$ is singular. It is usually represented as the pair (α, β), as there is a reasonable interpretation for $\beta = 0$ and even for both being zero.

The right generalized eigenvector $v(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies $A^*v(j) = \lambda(j)^*B^*v(j)$.

The left generalized eigenvector $u(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A,B) satisfies

$$u(j)^H A = \lambda(j)^* u(j)^H B$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$.

The ?ggev routine replaces the deprecated ?gegv routine.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>jobvl</i>	CHARACTER*1. Must be 'N' or 'V'.
--------------	----------------------------------

If $\text{jobvl} = \text{'N'}$, the left generalized eigenvectors are not computed;
 If $\text{jobvl} = \text{'V'}$, the left generalized eigenvectors are computed.

jobvr
 CHARACTER*1. Must be ' N ' or ' V '.
 If $\text{jobvr} = \text{'N'}$, the right generalized eigenvectors are not computed;
 If $\text{jobvr} = \text{'V'}$, the right generalized eigenvectors are computed.

n
 INTEGER. The order of the matrices A , B , vl , and vr ($n \geq 0$).

a , b , work
 REAL for sggev
 DOUBLE PRECISION for dggev
 COMPLEX for cggev
 DOUBLE COMPLEX for zggev.

Arrays:
 $a(\text{lda},*)$ is an array containing the n -by- n matrix A (first of the pair of matrices).
 The second dimension of a must be at least $\max(1, n)$.
 $b(\text{ldb},*)$ is an array containing the n -by- n matrix B (second of the pair of matrices).
 The second dimension of b must be at least $\max(1, n)$.
 work is a workspace array, its dimension $\max(1, \text{lwork})$.

lda
 INTEGER. The leading dimension of the array a . Must be at least $\max(1, n)$.

ldb
 INTEGER. The leading dimension of the array b . Must be at least $\max(1, n)$.

ldvl , ldvr
 INTEGER. The leading dimensions of the output matrices vl and vr , respectively.

Constraints:
 $\text{ldvl} \geq 1$. If $\text{jobvl} = \text{'V'}$, $\text{ldvl} \geq \max(1, n)$.
 $\text{ldvr} \geq 1$. If $\text{jobvr} = \text{'V'}$, $\text{ldvr} \geq \max(1, n)$.

lwork
 INTEGER.
 The dimension of the array work .
 $\text{lwork} \geq \max(1, 8n+16)$ for real flavors;
 $\text{lwork} \geq \max(1, 2n)$ for complex flavors.
 For good performance, lwork must generally be larger.
 If $\text{lwork} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the work array, returns this value as the first entry of the work array, and no error message related to lwork is issued by **xerbla**.

rwork
 REAL for cggev
 DOUBLE PRECISION for zggev

Workspace array, DIMENSION at least $\max(1, 8n)$.

This array is used in complex flavors only.

Output Parameters

a, b	On exit, these arrays have been overwritten.
$alphar, alphai$	<p>REAL for sggev;</p> <p>DOUBLE PRECISION for dggev.</p> <p>Arrays, DIMENSION at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.</p> <p>See <i>beta</i>.</p>
$alpha$	<p>COMPLEX for cggev;</p> <p>DOUBLE COMPLEX for zggev.</p> <p>Array, DIMENSION at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See <i>beta</i>.</p>
$beta$	<p>REAL for sggev</p> <p>DOUBLE PRECISION for dggev</p> <p>COMPLEX for cggev</p> <p>DOUBLE COMPLEX for zggev.</p> <p>Array, DIMENSION at least $\max(1, n)$.</p> <p><i>For real flavors:</i></p> <p>On exit, $(alphar(j) + alphai(j)*i)/beta(j)$, $j=1, \dots, n$, are the generalized eigenvalues.</p> <p>If $alphai(j)$ is zero, then the j-th eigenvalue is real; if positive, then the j-th and $(j+1)$-st eigenvalues are a complex conjugate pair, with $alphai(j+1)$ negative.</p> <p><i>For complex flavors:</i></p> <p>On exit, $alpha(j)/beta(j)$, $j=1, \dots, n$, are the generalized eigenvalues.</p> <p>See also <i>Application Notes</i> below.</p>
vl, vr	<p>REAL for sggev</p> <p>DOUBLE PRECISION for dggev</p> <p>COMPLEX for cggev</p> <p>DOUBLE COMPLEX for zggev.</p> <p>Arrays:</p> <p>$vl(Idvl, *)$; the second dimension of vl must be at least $\max(1, n)$.</p> <p>If $jobvl = 'V'$, the left generalized eigenvectors $u(j)$ are stored one after another in the columns of vl, in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.</p> <p>If $jobvl = 'N'$, vl is not referenced.</p>

For real flavors:

If the j-th eigenvalue is real, then $u(j) = v1(:,j)$, the j-th column of $v1$.

If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then $u(j) = v1(:,j) + i*v1(:,j+1)$ and $u(j+1) = v1(:,j) - i*v1(:,j+1)$, where $i = \sqrt{-1}$.

For complex flavors:

$u(j) = v1(:,j)$, the j-th column of $v1$.

$vr(lvdr,*);$ the second dimension of vr must be at least $\max(1, n)$.

If $jobvr = 'V'$, the right generalized eigenvectors $v(j)$ are stored one after another in the columns of vr , in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.

If $jobvr = 'N'$, vr is not referenced.

For real flavors:

If the j-th eigenvalue is real, then $v(j) = vr(:,j)$, the j-th column of vr .

If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then $v(j) = vr(:,j) + i*vr(:,j+1)$ and $v(j+1) = vr(:,j) - i*vr(:,j+1)$.

For complex flavors:

$v(j) = vr(:,j)$, the j-th column of vr .

$work(1)$

On exit, if $info = 0$, then $work(1)$ returns the required minimal size of $lwork$.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i$, and

$i \leq n$: the QZ iteration failed. No eigenvectors have been calculated, but $alphai(j)$, $alphai(j)$ (for real flavors), or $alpha(j)$ (for complex flavors), and $beta(j)$, $j=info+1, \dots, n$ should be correct.

$i > n$: errors that usually indicate LAPACK problems:

$i = n+1$: other than QZ iteration failed in [hgeqz](#);

$i = n+2$: error return from [tgevc](#).

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine [ggev](#) interface are the following:

a Holds the matrix A of size (n, n) .

b Holds the matrix B of size (n, n) .

<i>alphar</i>	Holds the vector of length n . Used in real flavors only.
<i>alphai</i>	Holds the vector of length n . Used in real flavors only.
<i>alpha</i>	Holds the vector of length n . Used in complex flavors only.
<i>beta</i>	Holds the vector of length n .
<i>vl</i>	Holds the matrix VL of size (n, n) .
<i>vr</i>	Holds the matrix VR of size (n, n) .
<i>jobvl</i>	Restored based on the presence of the argument <i>vl</i> as follows: <i>jobvl</i> = 'V', if <i>vl</i> is present, <i>jobvl</i> = 'N', if <i>vl</i> is omitted.
<i>jobvr</i>	Restored based on the presence of the argument <i>vr</i> as follows: <i>jobvr</i> = 'V', if <i>vr</i> is present, <i>jobvr</i> = 'N', if <i>vr</i> is omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar(j)/beta(j)* and *alphai(j)/beta(j)* may easily over- or underflow, and *beta(j)* may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with *norm(A)* in magnitude, and *beta* always less than and usually comparable with *norm(B)*.

?ggevx

Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.

Syntax

FORTRAN 77:

```
call sggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphar, alphai, beta, vl,
            ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work, lwork,
            iwork, bwork, info)

call dggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alphar, alphai, beta, vl,
            ldvl, vr, ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work, lwork,
            iwork, bwork, info)
```

```
call cggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr,
ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work, lwork, rwork,
iwork, bwork, info)

call zggevx(balanc, jobvl, jobvr, sense, n, a, lda, b, ldb, alpha, beta, vl, ldvl, vr,
ldvr, ilo, ihi, lscale, rscale, abnrm, bbnrm, rconde, rcondv, work, lwork, rwork,
iwork, bwork, info)
```

Fortran 95:

```
call ggevx(a, b, alphar, alphai, beta [,vl] [,vr] [,balanc] [,ilo] [,ihi] [, lscale]
[,rscale] [,abnrm] [,bbnrm] [,rconde] [,rcondv] [,info])

call ggevx(a, b, alpha, beta [, vl] [,vr] [,balanc] [,ilo] [,ihi] [,lscale] [, rscale]
[,abnrm] [,bbnrm] [,rconde] [,rcondv] [,info])
```

C:

```
lapack_int LAPACKE_sggevx( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float*
alphar, float* alphai, float* beta, float* vl, lapack_int ldvl, float* vr, lapack_int
ldvr, lapack_int* ilo, lapack_int* ihi, float* lscale, float* rscale, float* abnrm,
float* bbnrm, float* rconde, float* rcondv );

lapack_int LAPACKE_dggevx( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double*
alphar, double* alphai, double* beta, double* vl, lapack_int ldvl, double* vr,
lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, double* lscale, double* rscale,
double* abnrm, double* bbnrm, double* rconde, double* rcondv );

lapack_int LAPACKE_cggevx( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b,
lapack_int ldb, lapack_complex_float* alpha, lapack_complex_float* beta,
lapack_complex_float* vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr,
lapack_int* ilo, lapack_int* ihi, float* lscale, float* rscale, float* abnrm, float*
bbnrm, float* rconde, float* rcondv );

lapack_int LAPACKE_zggevx( int matrix_layout, char balanc, char jobvl, char jobvr, char
sense, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double*
b, lapack_int ldb, lapack_complex_double* alpha, lapack_complex_double* beta,
lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int ldvr,
lapack_int* ilo, lapack_int* ihi, double* lscale, double* rscale, double* abnrm,
double* bbnrm, double* rconde, double* rcondv );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine computes for a pair of n -by- n real/complex nonsymmetric matrices (A, B), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors ($ilo, ihi, lscale, rscale, abnrm$, and $bbnrm$), reciprocal condition numbers for the eigenvalues ($rconde$), and reciprocal condition numbers for the right eigenvectors ($rcondv$).

A generalized eigenvalue for a pair of matrices (A, B) is a scalar λ or a ratio $\text{alpha} / \text{beta} = \lambda$, such that $A - \lambda B$ is singular. It is usually represented as the pair $(\text{alpha}, \text{beta})$, as there is a reasonable interpretation for $\text{beta}=0$ and even for both being zero. The right generalized eigenvector $v(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A, B) satisfies

$$A^*v(j) = \lambda(j)^*B^*v(j).$$

The left generalized eigenvector $u(j)$ corresponding to the generalized eigenvalue $\lambda(j)$ of (A, B) satisfies

$$u(j)^H A = \lambda(j)^* u(j)^H B$$

where $u(j)^H$ denotes the conjugate transpose of $u(j)$.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>balanc</i>	CHARACTER*1. Must be 'N', 'P', 'S', or 'B'. Specifies the balance option to be performed. If <i>balanc</i> = 'N', do not diagonally scale or permute; If <i>balanc</i> = 'P', permute only; If <i>balanc</i> = 'S', scale only; If <i>balanc</i> = 'B', both permute and scale. Computed reciprocal condition numbers will be for the matrices after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.
<i>jobvl</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvl</i> = 'N', the left generalized eigenvectors are not computed; If <i>jobvl</i> = 'V', the left generalized eigenvectors are computed.
<i>jobvr</i>	CHARACTER*1. Must be 'N' or 'V'. If <i>jobvr</i> = 'N', the right generalized eigenvectors are not computed; If <i>jobvr</i> = 'V', the right generalized eigenvectors are computed.
<i>sense</i>	CHARACTER*1. Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed. If <i>sense</i> = 'N', none are computed; If <i>sense</i> = 'E', computed for eigenvalues only; If <i>sense</i> = 'V', computed for eigenvectors only; If <i>sense</i> = 'B', computed for eigenvalues and eigenvectors.
<i>n</i>	INTEGER. The order of the matrices A , B , vl , and vr ($n \geq 0$).
<i>a, b, work</i>	REAL for sggevx DOUBLE PRECISION for dggevx COMPLEX for cggevx DOUBLE COMPLEX for zggevx.

Arrays:

$a(la,*)$ is an array containing the n -by- n matrix A (first of the pair of matrices).

The second dimension of a must be at least $\max(1, n)$.

$b(lb,*)$ is an array containing the n -by- n matrix B (second of the pair of matrices).

The second dimension of b must be at least $\max(1, n)$.

$work$ is a workspace array, its dimension $\max(1, lwork)$.

la

INTEGER. The leading dimension of the array a .

Must be at least $\max(1, n)$.

lb

INTEGER. The leading dimension of the array b .

Must be at least $\max(1, n)$.

$ldvl, ldvr$

INTEGER. The leading dimensions of the output matrices vl and vr , respectively.

Constraints:

$ldvl \geq 1$. If $jobvl = 'V'$, $ldvl \geq \max(1, n)$.

$ldvr \geq 1$. If $jobvr = 'V'$, $ldvr \geq \max(1, n)$.

$lwork$

INTEGER.

The dimension of the array $work$. $lwork \geq \max(1, 2*n)$;

For real flavors:

If $balanc = 'S'$, or $'B'$, or $jobvl = 'V'$, or $jobvr = 'V'$, then $lwork \geq \max(1, 6*n)$;

if $sense = 'E'$, or $'B'$, then $lwork \geq \max(1, 10*n)$;

if $sense = 'V'$, or $'B'$, $lwork \geq (2n^2 + 8*n + 16)$.

For complex flavors:

if $sense = 'E'$, $lwork \geq \max(1, 4*n)$;

if $sense = 'V'$, or $'B'$, $lwork \geq \max(1, 2*n^2 + 2*n)$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [xerbla](#).

$rwork$

REAL for cggevx

DOUBLE PRECISION for zggevx

Workspace array, DIMENSION at least $\max(1, 6*n)$ if $balanc = 'S'$, or $'B'$, and at least $\max(1, 2*n)$ otherwise.

This array is used in complex flavors only.

$iwork$

INTEGER.

Workspace array, DIMENSION at least $(n+6)$ for real flavors and at least $(n+2)$ for complex flavors.

Not referenced if $sense = 'E'$.

bwork

LOGICAL. Workspace array, DIMENSION at least $\max(1, n)$.

Not referenced if $sense = 'N'$.

Output Parameters

a, b

On exit, these arrays have been overwritten.

If $jobvl = 'V'$ or $jobvr = 'V'$ or both, then *a* contains the first part of the real Schur form of the "balanced" versions of the input *A* and *B*, and *b* contains its second part.

alphar, alphai

REAL for sggevx;

DOUBLE PRECISION for dggevx.

Arrays, DIMENSION at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.

See *beta*.

alpha

COMPLEX for cggevx;

DOUBLE COMPLEX for zggevx.

Array, DIMENSION at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See *beta*.

beta

REAL for sggevx

DOUBLE PRECISION for dggevx

COMPLEX for cggevx

DOUBLE COMPLEX for zggevx.

Array, DIMENSION at least $\max(1, n)$.

For real flavors:

On exit, $(\text{alphar}(j) + \text{alphai}(j)*i)/\text{beta}(j)$, $j=1, \dots, n$, will be the generalized eigenvalues.

If $\text{alphai}(j)$ is zero, then the j -th eigenvalue is real; if positive, then the j -th and $(j+1)$ -st eigenvalues are a complex conjugate pair, with $\text{alphai}(j+1)$ negative.

For complex flavors:

On exit, $\text{alpha}(j)/\text{beta}(j)$, $j=1, \dots, n$, will be the generalized eigenvalues.

See also Application Notes below.

vl, vr

REAL for sggevx

DOUBLE PRECISION for dggevx

COMPLEX for cggevx

DOUBLE COMPLEX for zggevx.

Arrays:

$vl(lv1,*)$; the second dimension of vl must be at least $\max(1, n)$.

If $jobvl = 'V'$, the left generalized eigenvectors $u(j)$ are stored one after another in the columns of vl , in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.

If $jobvl = 'N'$, vl is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $u(j) = vl(:,j)$, the j -th column of vl .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $u(j) = vl(:,j) + i * vl(:,j+1)$ and $u(j+1) = vl(:,j) - i * vl(:,j+1)$, where $i = \sqrt{-1}$.

For complex flavors:

$u(j) = vl(:,j)$, the j -th column of vl .

$vr(lv1,*)$; the second dimension of vr must be at least $\max(1, n)$.

If $jobvr = 'V'$, the right generalized eigenvectors $v(j)$ are stored one after another in the columns of vr , in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have $\text{abs}(\text{Re}) + \text{abs}(\text{Im}) = 1$.

If $jobvr = 'N'$, vr is not referenced.

For real flavors:

If the j -th eigenvalue is real, then $v(j) = vr(:,j)$, the j -th column of vr .

If the j -th and $(j+1)$ -st eigenvalues form a complex conjugate pair, then $v(j) = vr(:,j) + i * vr(:,j+1)$ and $v(j+1) = vr(:,j) - i * vr(:,j+1)$.

For complex flavors:

$v(j) = vr(:,j)$, the j -th column of vr .

ilo, ihi

INTEGER. ilo and ihi are integer values such that on exit $A(i,j) = 0$ and $B(i,j) = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ihi+1, \dots, n$.

If $balanc = 'N'$ or $'S'$, $ilo = 1$ and $ihi = n$.

$lscale, rscale$

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Arrays, DIMENSION at least $\max(1, n)$ each.

$lscale$ contains details of the permutations and scaling factors applied to the left side of A and B .

If $PL(j)$ is the index of the row interchanged with row j , and $DL(j)$ is the scaling factor applied to row j , then

```

 $lscale(j) = PL(j), \text{ for } j = 1, \dots, ilo-1$ 
 $= DL(j), \text{ for } j = ilo, \dots, ihi$ 
 $= PL(j) \text{ for } j = ihi+1, \dots, n.$ 

```

The order in which the interchanges are made is n to $ihi+1$, then 1 to $ilo-1$.

rscale contains details of the permutations and scaling factors applied to the right side of *A* and *B*.

If *PR(j)* is the index of the column interchanged with column *j*, and *DR(j)* is the scaling factor applied to column *j*, then

$$\begin{aligned} \text{rscale}(j) &= \text{PR}(j), \text{ for } j = 1, \dots, \text{ilo}-1 \\ &= \text{DR}(j), \text{ for } j = \text{ilo}, \dots, \text{ihi} \\ &= \text{PR}(j) \text{ for } j = \text{ihi}+1, \dots, n. \end{aligned}$$

The order in which the interchanges are made is *n* to *ihi*+1, then 1 to *ilo*-1.

abnrm, *bbnrm*

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

The one-norms of the balanced matrices *A* and *B*, respectively.

rconde, *rcondv*

REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

Arrays, DIMENSION at least max(1, *n*) each.

If *sense* = 'E', or 'B', *rconde* contains the reciprocal condition numbers of the eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of *rconde* are set to the same value. Thus *rconde(j)*, *rcondv(j)*, and the *j*-th columns of *vl* and *vr* all correspond to the same eigenpair (but not in general the *j*-th eigenpair, unless all eigenpairs are selected).

If *sense* = 'N', or 'V', *rconde* is not referenced.

If *sense* = 'V', or 'B', *rcondv* contains the estimated reciprocal condition numbers of the eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of *rcondv* are set to the same value.

If the eigenvalues cannot be reordered to compute *rcondv(j)*, *rcondv(j)* is set to 0; this can only occur when the true value would be very small anyway.

If *sense* = 'N', or 'E', *rcondv* is not referenced.

work(1)

On exit, if *info* = 0, then *work(1)* returns the required minimal size of *lwork*.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = -*i*, the *i*-th parameter had an illegal value.

If *info* = *i*, and

i ≤ *n*:

the QZ iteration failed. No eigenvectors have been calculated, but *alphar(j)*, *alphai(j)* (for real flavors), or *alpha(j)* (for complex flavors), and *beta(j)*, *j*=*info*+1, ..., *n* should be correct.

i > *n*: errors that usually indicate LAPACK problems:

i = *n*+1: other than QZ iteration failed in [hgeqz](#);

i = *n*+2: error return from [tgevc](#).

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or restorable arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `ggevx` interface are the following:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n</i> , <i>n</i>).
<i>b</i>	Holds the matrix <i>B</i> of size (<i>n</i> , <i>n</i>).
<i>alphar</i>	Holds the vector of length <i>n</i> . Used in real flavors only.
<i>alphaai</i>	Holds the vector of length <i>n</i> . Used in real flavors only.
<i>alpha</i>	Holds the vector of length <i>n</i> . Used in complex flavors only.
<i>beta</i>	Holds the vector of length <i>n</i> .
<i>vl</i>	Holds the matrix <i>VL</i> of size (<i>n</i> , <i>n</i>).
<i>vr</i>	Holds the matrix <i>VR</i> of size (<i>n</i> , <i>n</i>).
<i>lscale</i>	Holds the vector of length <i>n</i> .
<i>rscale</i>	Holds the vector of length <i>n</i> .
<i>rconde</i>	Holds the vector of length <i>n</i> .
<i>rcondv</i>	Holds the vector of length <i>n</i> .
<i>balanc</i>	Must be 'N', 'B', or 'P'. The default value is 'N'.
<i>jobvl</i>	Restored based on the presence of the argument <i>vl</i> as follows: <i>jobvl</i> = 'V', if <i>vl</i> is present, <i>jobvl</i> = 'N', if <i>vl</i> is omitted.
<i>jobvr</i>	Restored based on the presence of the argument <i>vr</i> as follows: <i>jobvr</i> = 'V', if <i>vr</i> is present, <i>jobvr</i> = 'N', if <i>vr</i> is omitted.
<i>sense</i>	Restored based on the presence of arguments <i>rconde</i> and <i>rcondv</i> as follows: <i>sense</i> = 'B', if both <i>rconde</i> and <i>rcondv</i> are present, <i>sense</i> = 'E', if <i>rconde</i> is present and <i>rcondv</i> omitted, <i>sense</i> = 'V', if <i>rconde</i> is omitted and <i>rcondv</i> present, <i>sense</i> = 'N', if both <i>rconde</i> and <i>rcondv</i> are omitted.

Application Notes

If you are in doubt how much workspace to supply, use a generous value of *lwork* for the first run or set *lwork* = -1.

If you choose the first option and set any of admissible *lwork* sizes, which is no less than the minimal value described, the routine completes the task, though probably not so fast as with a recommended workspace, and provides the recommended workspace in the first element of the corresponding array *work* on exit. Use this value (*work*(1)) for subsequent runs.

If you set *lwork* = -1, the routine returns immediately and provides the recommended workspace in the first element of the corresponding array (*work*). This operation is called a workspace query.

Note that if you set *lwork* to less than the minimal required value and not -1, the routine returns immediately with an error exit and does not provide any information on the recommended workspace.

The quotients *alphar(j)/beta(j)* and *alphai(j)/beta(j)* may easily over- or underflow, and *beta(j)* may even be zero. Thus, you should avoid simply computing the ratio. However, *alphar* and *alphai* (for real flavors) or *alpha* (for complex flavors) will be always less than and usually comparable with *norm(A)* in magnitude, and *beta* always less than and usually comparable with *norm(B)*.

Auxiliary Routines

Routine naming conventions, mathematical notation, and matrix storage schemes used for LAPACK auxiliary routines are the same as for the driver and computational routines described in previous chapters.

The table below summarizes information about the available LAPACK auxiliary routines.

LAPACK Auxiliary Routines

Routine Name	Data Types	Description
?lacgv	c, z	Conjugates a complex vector.
?lacrm	c, z	Multiplies a complex matrix by a square real matrix.
?lacrt	c, z	Performs a linear transformation of a pair of complex vectors.
?laesy	c, z	Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix.
?rot	c, z	Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.
?spmv	c, z	Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix
?spr	c, z	Performs the symmetrical rank-1 update of a complex symmetric packed matrix.
?syconv	s, c, d, z	Converts a symmetric matrix given by a triangular matrix factorization into two matrices and vice versa.
?symv	c, z	Computes a matrix-vector product for a complex symmetric matrix.
?syr	c, z	Performs the symmetric rank-1 update of a complex symmetric matrix.
i?max1	c, z	Finds the index of the vector element whose real part has maximum absolute value.
?sum1	sc, dz	Forms the 1-norm of the complex vector using the true absolute value.
?gbtf2	s, d, c, z	Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.

Routine Name	Data Types	Description
?gebd2	s, d, c, z	Reduces a general matrix to bidiagonal form using an unblocked algorithm.
?gehd2	s, d, c, z	Reduces a general square matrix to upper Hessenberg form using an unblocked algorithm.
?gelq2	s, d, c, z	Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.
?geql2	s, d, c, z	Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.
?geqr2	s, d, c, z	Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.
?geqr2p	s, d, c, z	Computes the QR factorization of a general rectangular matrix with non-negative diagonal elements using an unblocked algorithm.
?geqrt2	s, d, c, z	Computes a QR factorization of a general real or complex matrix using the compact WY representation of Q.
?geqrt3	s, d, c, z	Recursively computes a QR factorization of a general real or complex matrix using the compact WY representation of Q.
?gerq2	s, d, c, z	Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.
?gesc2	s, d, c, z	Solves a system of linear equations using the LU factorization with complete pivoting computed by ?getc2.
?getc2	s, d, c, z	Computes the LU factorization with complete pivoting of the general n -by- n matrix.
?getf2	s, d, c, z	Computes the LU factorization of a general m -by- n matrix using partial pivoting with row interchanges (unblocked algorithm).
?gtts2	s, d, c, z	Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf.
?isnan	s, d,	Tests input for NaN.
?laisnan	s, d,	Tests input for NaN by comparing two arguments for inequality.
?labrd	s, d, c, z	Reduces the first nb rows and columns of a general matrix to a bidiagonal form.
?lacn2	s, d, c, z	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
?lacon	s, d, c, z	Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.
?lacpy	s, d, c, z	Copies all or part of one two-dimensional array to another.
?ladiv	s, d, c, z	Performs complex division in real arithmetic, avoiding unnecessary overflow.
?lae2	s, d	Computes the eigenvalues of a 2-by-2 symmetric matrix.
?laebz	s, d	Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine ?stebz.

Routine Name	Data Types	Description
?laed0	s, d, c, z	Used by ?stedc. Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.
?laed1	s, d	Used by ssstedc/dstedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.
?laed2	s, d	Used by ssstedc/dstedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.
?laed3	s, d	Used by ssstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.
?laed4	s, d	Used by ssstedc/dstedc. Finds a single root of the secular equation.
?laed5	s, d	Used by ssstedc/dstedc. Solves the 2-by-2 secular equation.
?laed6	s, d	Used by ssstedc/dstedc. Computes one Newton step in solution of the secular equation.
?laed7	s, d, c, z	Used by ?stedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.
?laed8	s, d, c, z	Used by ?stedc. Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.
?laed9	s, d	Used by ssstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.
?laeda	s, d	Used by ?stedc. Computes the Z vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.
?laein	s, d, c, z	Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.
?laev2	s, d, c, z	Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.
?laexc	s, d	Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.
?lag2	s, d	Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.
?lags2	s, d	Computes 2-by-2 orthogonal matrices U , V , and Q , and applies them to matrices A and B such that the rows of the transformed A and B are parallel.
?lagtf	s, d	Computes an LU factorization of a matrix $T - \lambda I$, where T is a general tridiagonal matrix, and λ a scalar, using partial pivoting with row interchanges.
?lagtm	s, d, c, z	Performs a matrix-matrix product of the form $C = \alpha AB + \beta C$, where A is a tridiagonal matrix, B and C are rectangular matrices, and α and β are scalars, which may be 0, 1, or -1.

Routine Name	Data Types	Description
?lagts	s, d	Solves the system of equations $(T - \lambda I)x = y$ or $(T - \lambda I)^T x = y$, where T is a general tridiagonal matrix and λ a scalar, using the LU factorization computed by ?lagtf.
?lagv2	s, d	Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (A, B) where B is upper triangular.
?lahqr	s, d, c, z	Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.
?lahrd	s, d, c, z	Reduces the first nb columns of a general rectangular matrix A so that elements below the k -th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .
?lahr2	s, d, c, z	Reduces the specified number of first columns of a general rectangular matrix A so that elements below the specified subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .
?laic1	s, d, c, z	Applies one step of incremental condition estimation.
?laln2	s, d	Solves a 1-by-1 or 2-by-2 linear system of equations of the specified form.
?lals0	s, d, c, z	Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by ?gelsd.
?lalsa	s, d, c, z	Computes the SVD of the coefficient matrix in compact form. Used by ?gelsd.
?lalsd	s, d, c, z	Uses the singular value decomposition of A to solve the least squares problem.
?lamrg	s, d	Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in ascending order.
?laneg	s, d	Computes the Sturm count.
?langb	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.
?lange	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.
?langt	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.
?lanhs	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.
?lansb	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.

Routine Name	Data Types	Description
?lanhb	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.
?lansp	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.
?lanhp	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.
?lanst/?lanht	s, d/c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.
?lansy	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.
?lanhe	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.
?lantb	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.
?lantp	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.
?lantr	s, d, c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.
?lanv2	s, d	Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.
?lapll	s, d, c, z	Measures the linear dependence of two vectors.
?lapmr	s, d, c, z	Rearranges rows of a matrix as specified by a permutation vector.
?lapmt	s, d, c, z	Performs a forward or backward permutation of the columns of a matrix.
?lapy2	s, d	Returns $\sqrt{x^2+y^2}$.
?lapy3	s, d	Returns $\sqrt{x^2+y^2+z^2}$.
?laqgb	s, d, c, z	Scales a general band matrix, using row and column scaling factors computed by ?gbequ.
?laqge	s, d, c, z	Scales a general rectangular matrix, using row and column scaling factors computed by ?geequ.
?laqhb	c, z	Scales a Hermitian band matrix, using scaling factors computed by ?pbequ.
?laqp2	s, d, c, z	Computes a QR factorization with column pivoting of the matrix block.

Routine Name	Data Types	Description
?laqps	s, d, c, z	Computes a step of QR factorization with column pivoting of a real m-by-n matrix A by using BLAS level 3.
?laqr0	s, d, c, z	Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.
?laqr1	s, d, c, z	Sets a scalar multiple of the first column of the product of 2-by-2 or 3-by-3 matrix H and specified shifts.
?laqr2	s, d, c, z	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
?laqr3	s, d, c, z	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
?laqr4	s, d, c, z	Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.
?laqr5	s, d, c, z	Performs a single small-bulge multi-shift QR sweep.
?laqsb	s, d, c, z	Scales a symmetric/Hermitian band matrix, using scaling factors computed by ?pbequ.
?laqsp	s, d, c, z	Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by ?ppequ.
?laqsy	s, d, c, z	Scales a symmetric/Hermitian matrix, using scaling factors computed by ?poequ.
?laqtr	s, d	Solves a real quasi-triangular system of equations, or a complex quasi-triangular system of special form, in real arithmetic.
?lar1v	s, d, c, z	Computes the (scaled) r -th column of the inverse of the submatrix in rows b_1 through b_n of the tridiagonal matrix $LDL^T - \lambda I$.
?lar2v	s, d, c, z	Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.
?larf	s, d, c, z	Applies an elementary reflector to a general rectangular matrix.
?larfb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
?larfg	s, d, c, z	Generates an elementary reflector (Householder matrix).
?larfgp	s, d, c, z	Generates an elementary reflector (Householder matrix) with non-negative beta.
?larft	s, d, c, z	Forms the triangular factor T of a block reflector $H = I - vtv^H$
?larfx	s, d, c, z	Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order ≤ 10 .
?largv	s, d, c, z	Generates a vector of plane rotations with real cosines and real/complex sines.
?larnv	s, d, c, z	Returns a vector of random numbers from a uniform or normal distribution.

Routine Name	Data Types	Description
?larra	s, d	Computes the splitting points with the specified threshold.
?larrb	s, d	Provides limited bisection to locate eigenvalues for more accuracy.
?larrc	s, d	Computes the number of eigenvalues of the symmetric tridiagonal matrix.
?larrd	s, d	Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.
?larre	s, d	Given the tridiagonal matrix T , sets small off-diagonal elements to zero and for each unreduced block T_i , finds base representations and eigenvalues.
?larrf	s, d	Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.
?larrj	s, d	Performs refinement of the initial estimates of the eigenvalues of the matrix T .
?larrk	s, d	Computes one eigenvalue of a symmetric tridiagonal matrix T to suitable accuracy.
?larrr	s, d	Performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.
?larrv	s, d, c, z	Computes the eigenvectors of the tridiagonal matrix $T = L D L^T$ given L , D and the eigenvalues of $L D L^T$.
?lartg	s, d, c, z	Generates a plane rotation with real cosine and real/complex sine.
?lartgp	s, d	Generates a plane rotation so that the diagonal is nonnegative.
?lartgs	s, d	Generates a plane rotation designed to introduce a bulge in implicit QR iteration for the bidiagonal SVD problem.
?lartv	s, d, c, z	Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.
?laruv	s, d	Returns a vector of n random real numbers from a uniform distribution.
?larz	s, d, c, z	Applies an elementary reflector (as returned by ?tzrzf) to a general matrix.
?larzb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose to a general matrix.
?larzt	s, d, c, z	Forms the triangular factor T of a block reflector $H = I - vtv^H$.
?las2	s, d	Computes singular values of a 2-by-2 triangular matrix.
?lascl	s, d, c, z	Multiplies a general rectangular matrix by a real scalar defined as c_{to}/c_{from} .
?lasd0	s, d	Computes the singular values of a real upper bidiagonal n-by-m matrix B with diagonal d and off-diagonal e . Used by ?bdsdc.
?lasd1	s, d	Computes the SVD of an upper bidiagonal matrix B of the specified size. Used by ?bdsdc.

Routine Name	Data Types	Description
?lasd2	s, d	Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.
?lasd3	s, d	Finds all square roots of the roots of the secular equation, as defined by the values in D and Z , and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.
?lasd4	s, d	Computes the square root of the i -th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by ?bdsdc.
?lasd5	s, d	Computes the square root of the i -th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix. Used by ?bdsdc.
?lasd6	s, d	Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc.
?lasd7	s, d	Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.
?lasd8	s, d	Finds the square roots of the roots of the secular equation, and stores, for each element in D , the distance to its two nearest poles. Used by ?bdsdc.
?lasda	s, d	Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal d and off-diagonal e . Used by ?bdsdc.
?lasdq	s, d	Computes the SVD of a real bidiagonal matrix with diagonal d and off-diagonal e . Used by ?bdsdc.
?lasdt	s, d	Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.
?laset	s, d, c, z	Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.
?lasq1	s, d	Computes the singular values of a real square bidiagonal matrix. Used by ?bdsqr.
?lasq2	s, d	Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the qd Array Z to high relative accuracy. Used by ?bdsqr and ?stegr.
?lasq3	s, d	Checks for deflation, computes a shift and calls $dqds$. Used by ?bdsqr.
?lasq4	s, d	Computes an approximation to the smallest eigenvalue using values of d from the previous transform. Used by ?bdsqr.
?lasq5	s, d	Computes one $dqds$ transform in ping-pong form. Used by ?bdsqr and ?stegr.
?lasq6	s, d	Computes one dqd transform in ping-pong form. Used by ?bdsqr and ?stegr.
?lasr	s, d, c, z	Applies a sequence of plane rotations to a general rectangular matrix.

Routine Name	Data Types	Description
?lasrt	s, d	Sorts numbers in increasing or decreasing order.
?lassq	s, d, c, z	Updates a sum of squares represented in scaled form.
?lasv2	s, d	Computes the singular value decomposition of a 2-by-2 triangular matrix.
?laswp	s, d, c, z	Performs a series of row interchanges on a general rectangular matrix.
?lasy2	s, d	Solves the Sylvester matrix equation where the matrices are of order 1 or 2.
?lasyf	s, d, c, z	Computes a partial factorization of a real/complex symmetric matrix, using the diagonal pivoting method.
?lasyf_rook	s, d, c, z	Computes a partial factorization of a real/complex symmetric matrix, using the bounded Bunch-Kaufman diagonal pivoting method.
?lahef	c, z	Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.
?lahef_rook	c, z	Computes a partial factorization of a complex Hermitian indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method.
?latbs	s, d, c, z	Solves a triangular banded system of equations.
?latdf	s, d, c, z	Uses the LU factorization of the n -by- n matrix computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate.
?latps	s, d, c, z	Solves a triangular system of equations with the matrix held in packed storage.
?latrd	s, d, c, z	Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.
?latrs	s, d, c, z	Solves a triangular system of equations with the scale factor set to prevent overflow.
?latrz	s, d, c, z	Factors an upper trapezoidal matrix by means of orthogonal/unitary transformations.
?laau2	s, d, c, z	Computes the product UU^H or L^HL , where U and L are upper or lower triangular matrices (unblocked algorithm).
?laum	s, d, c, z	Computes the product UU^H or L^HL , where U and L are upper or lower triangular matrices (blocked algorithm).
?orbdb1/?unbdb1	s, d, c, z	Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.
?orbdb2/?unbdb2		
?orbdb3/?unbdb3		
?orbdb4/?unbdb4		
?orbdb5/?unbdb5	s, d, c, z	Orthogonalizes a column vector with respect to the orthonormal basis matrix.
?orbdb6/?unbdb6		

Routine Name	Data Types	Description
?org2l/?ung2l	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by ?geqlf (unblocked algorithm).
?org2r/?ung2r	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by ?geqrf (unblocked algorithm).
?orgl2/?ungl2	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by ?gelqf (unblocked algorithm).
?orgr2/?ungr2	s, d/c, z	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by ?gerqf (unblocked algorithm).
?orm2l/?unm2l	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf (unblocked algorithm).
?orm2r/?unm2r	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by ?geqrf (unblocked algorithm).
?orml2/?unml2	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by ?gelqf (unblocked algorithm).
?ormr2/?unmr2	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by ?gerqf (unblocked algorithm).
?ormr3/?unmr3	s, d/c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by ?tzrzf (unblocked algorithm).
?pbtf2	s, d, c, z	Computes the Cholesky factorization of a symmetric/ Hermitian positive definite band matrix (unblocked algorithm).
?potf2	s, d, c, z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (unblocked algorithm).
?ptts2	s, d, c, z	Solves a tridiagonal system of the form $AX=B$ using the $L D L^H$ factorization computed by ?pttrf.
?rscl	s, d, cs, zd	Multiplies a vector by the reciprocal of a real scalar.
?syswapr	s, d, c, z	Applies an elementary permutation on the rows and columns of a symmetric matrix.
?heswapr	c, z	Applies an elementary permutation on the rows and columns of a Hermitian matrix.
?sygs2/?hegs2	s, d/c, z	Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from ?potrf (unblocked algorithm).
?sytd2/?hetd2	s, d/c, z	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (unblocked algorithm).
?sytf2	s, d, c, z	Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).
?sytf2_rook	s, d, c, z	Computes the factorization of a real/complex symmetric indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).
?hetf2	c, z	Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).

Routine Name	Data Types	Description
?hetf2_rook	c, z	Computes the factorization of a complex Hermitian matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).
?tgex2	s, d, c, z	Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.
?tgsy2	s, d, c, z	Solves the generalized Sylvester equation (unblocked algorithm).
?trti2	s, d, c, z	Computes the inverse of a triangular matrix (unblocked algorithm).
clag2z	c → z	Converts a complex single precision matrix to a complex double precision matrix.
dlag2s	d → s	Converts a double precision matrix to a single precision matrix.
slag2d	s → d	Converts a single precision matrix to a double precision matrix.
zlag2c	z → c	Converts a complex double precision matrix to a complex single precision matrix.
?larfp	s, d, c, z	Generates a real or complex elementary reflector.
ila?lc	s, d, c, z	Scans a matrix for its last non-zero column.
ila?lr	s, d, c, z	Scans a matrix for its last non-zero row.
?gsvj0	s, d	Pre-processor for the routine ?gesvj.
?gsvj1	s, d	Pre-processor for the routine ?gesvj, applies Jacobi rotations targeting only particular pivots.
?sfrk	s, d	Performs a symmetric rank-k operation for matrix in RFP format.
?hfrk	c, z	Performs a Hermitian rank-k operation for matrix in RFP format.
?tfsm	s, d, c, z	Solves a matrix equation (one operand is a triangular matrix in RFP format).
?lansf	s, d	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix in RFP format.
?lanhf	c, z	Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian matrix in RFP format.
?tfttp	s, d, c, z	Copies a triangular matrix from the rectangular full packed format (TF) to the standard packed format (TP).
?tftrr	s, d, c, z	Copies a triangular matrix from the rectangular full packed format (TF) to the standard full format (TR).
?tpqrt2	s, d, c, z	Computes a QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for Q.
?tprfb	s, d, c, z	Applies a real or complex "triangular-pentagonal" blocked reflector to a real or complex matrix, which is composed of two blocks.
?tpptf	s, d, c, z	Copies a triangular matrix from the standard packed format (TP) to the rectangular full packed format (TF).

Routine Name	Data Types	Description
?tpstr	s, d, c, z	Copies a triangular matrix from the standard packed format (TP) to the standard full format (TR).
?trtf	s, d, c, z	Copies a triangular matrix from the standard full format (TR) to the rectangular full packed format (TF).
?trtp	s, d, c, z	Copies a triangular matrix from the standard full format (TR) to the standard packed format (TP).
?pstf2	s, d, c, z	Computes the Cholesky factorization with complete pivoting of a real symmetric or complex Hermitian positive semi-definite matrix.
dlat2s	d → s	Converts a double-precision triangular matrix to a single-precision triangular matrix.
zlat2c	z → c	Converts a double complex triangular matrix to a complex triangular matrix.
?lacp2	c, z	Copies all or part of a real two-dimensional array to a complex array.
?la_gbamv	s, d, c, z	Performs a matrix-vector operation to calculate error bounds.
?la_gbrcond	s, d	Estimates the Skeel condition number for a general banded matrix.
?la_gbrcond_c	c, z	Computes the infinity norm condition number of $\text{op}(A)*\text{inv}(\text{diag}(c))$ for general banded matrices.
?la_gbrcond_x	c, z	Computes the infinity norm condition number of $\text{op}(A)*\text{diag}(x)$ for general banded matrices.
?la_gbrfsx_extended	s, d, c, z	Improves the computed solution to a system of linear equations for general banded matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_gbrpvgrw	s, d, c, z	Computes the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$ for a general banded matrix.
?la_geamv	s, d, c, z	Computes a matrix-vector product using a general matrix to calculate error bounds.
?la_gercond	s, d	Estimates the Skeel condition number for a general matrix.
?la_gercond_c	c, z	Computes the infinity norm condition number of $\text{op}(A)*\text{inv}(\text{diag}(c))$ for general matrices.
?la_gercond_x	c, z	Computes the infinity norm condition number of $\text{op}(A)*\text{diag}(x)$ for general matrices.
?la_gerfsx_extended	s, d	Improves the computed solution to a system of linear equations for general matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_heamv	c, z	Computes a matrix-vector product using a Hermitian indefinite matrix to calculate error bounds.
?la_hercond_c	c, z	Computes the infinity norm condition number of $\text{op}(A)*\text{inv}(\text{diag}(c))$ for Hermitian indefinite matrices.

Routine Name	Data Types	Description
?la_hercond_x	c, z	Computes the infinity norm condition number of $\text{op}(A) * \text{diag}(x)$ for Hermitian indefinite matrices.
?la_herfsx_extended	c, z	Improves the computed solution to a system of linear equations for Hermitian indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_lin_berr	s, d, c, z	Computes a component-wise relative backward error.
?la_porcond	s, d	Estimates the Skeel condition number for a symmetric positive-definite matrix.
?la_porcond_c	c, z	Computes the infinity norm condition number of $\text{op}(A) * \text{inv}(\text{diag}(c))$ for Hermitian positive-definite matrices.
?la_porcond_x	c, z	Computes the infinity norm condition number of $\text{op}(A) * \text{diag}(x)$ for Hermitian positive-definite matrices.
?la_porfsx_extended	s, d, c, z	Improves the computed solution to a system of linear equations for symmetric or Hermitian positive-definite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_porpvgrow	s, d, c, z	Computes the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$ for a symmetric or Hermitian positive-definite matrix.
?laqhe	c, z	Scales a Hermitian matrix.
?laqhp	c, z	Scales a Hermitian matrix stored in packed form.
?larcm	c, z	Copies all or part of a real two-dimensional array to a complex array.
?la_rpvgrw	c, z	Multiplies a square real matrix by a complex matrix.
?larscl2	s, d, c, z	Performs reciprocal diagonal scaling on a vector.
?lascl2	s, d, c, z	Performs diagonal scaling on a vector.
?la_syamv	s, d, c, z	Computes a matrix-vector product using a symmetric indefinite matrix to calculate error bounds.
?la_syrcond	s, d	Estimates the Skeel condition number for a symmetric indefinite matrix.
?la_syrcond_c	c, z	Computes the infinity norm condition number of $\text{op}(A) * \text{inv}(\text{diag}(c))$ for symmetric indefinite matrices.
?la_syrcond_x	c, z	Computes the infinity norm condition number of $\text{op}(A) * \text{diag}(x)$ for symmetric indefinite matrices.
?la_syrfpsx_extended	s, d, c, z	Improves the computed solution to a system of linear equations for symmetric indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.
?la_syrvgrw	s, d, c, z	Computes the reciprocal pivot growth factor $\text{norm}(A)/\text{norm}(U)$ for a symmetric indefinite matrix.
?la_wwaddw	s, d, c, z	Adds a vector into a doubled-single vector.

?lacgv*Conjugates a complex vector.***Syntax**

```
call clacgv( n, x, incx )
```

```
call zlacgv( n, x, incx )
```

C:

```
lapack_int LAPACKE_<?>lacgv (lapack_int n, <datatype> * x, lapack_int incx);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine conjugates a complex vector x of length n and increment $incx$ (see "Vector Arguments in BLAS" in Appendix B).

Input Parameters

The data types are given for the Fortran interface.

n INTEGER. The length of the vector x ($n \geq 0$).

x COMPLEX for clacgv
 DOUBLE COMPLEX for zlacgv.
 Array, dimension $(1+(n-1) * |incx|)$.
 Contains the vector of length n to be conjugated.

incx INTEGER. The spacing between successive elements of x .

Output Parameters

x On exit, overwritten with $\text{conjg}(x)$.

?lacrm*Multiples a complex matrix by a square real matrix.***Syntax**

```
call clacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

```
call zlacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine performs a simple matrix-matrix multiplication of the form

$C = A^*B$,

where A is m -by- n and complex, B is n -by- n and real, C is m -by- n and complex.

Input Parameters

m	INTEGER. The number of rows of the matrix A and of the matrix C ($m \geq 0$).
n	INTEGER. The number of columns and rows of the matrix B and the number of columns of the matrix C ($n \geq 0$).
a	COMPLEX for <code>clacrm</code> DOUBLE COMPLEX for <code>zlacrm</code> Array, DIMENSION (lda, n). Contains the m -by- n matrix A .
lda	INTEGER. The leading dimension of the array a , $lda \geq \max(1, m)$.
b	REAL for <code>clacrm</code> DOUBLE PRECISION for <code>zlacrm</code> Array, DIMENSION (ldb, n). Contains the n -by- n matrix B .
ldb	INTEGER. The leading dimension of the array b , $ldb \geq \max(1, n)$.
ldc	INTEGER. The leading dimension of the output array c , $ldc \geq \max(1, n)$.
$rwork$	REAL for <code>clacrm</code> DOUBLE PRECISION for <code>zlacrm</code> Workspace array, DIMENSION ($2*m*n$).

Output Parameters

c	COMPLEX for <code>clacrm</code> DOUBLE COMPLEX for <code>zlacrm</code> Array, DIMENSION (ldc, n). Contains the m -by- n matrix C .
-----	--

?lacrt

Performs a linear transformation of a pair of complex vectors.

Syntax

```
call clacrt(  $n$ ,  $cx$ ,  $incx$ ,  $cy$ ,  $incy$ ,  $c$ ,  $s$  )
call zlacrt(  $n$ ,  $cx$ ,  $incx$ ,  $cy$ ,  $incy$ ,  $c$ ,  $s$  )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine performs the following transformation

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \Rightarrow \begin{pmatrix} x \\ y \end{pmatrix},$$

where c, s are complex scalars and x, y are complex vectors.

Input Parameters

n	INTEGER. The number of elements in the vectors cx and cy ($n \geq 0$).
cx, cy	COMPLEX for <code>clacrt</code> DOUBLE COMPLEX for <code>zlacrt</code> Arrays, dimension (n). Contain input vectors x and y , respectively.
$incx$	INTEGER. The increment between successive elements of cx .
$incy$	INTEGER. The increment between successive elements of cy .
c, s	COMPLEX for <code>clacrt</code> DOUBLE COMPLEX for <code>zlacrt</code> Complex scalars that define the transform matrix

|
|

Output Parameters

cx	On exit, overwritten with $c*x + s*y$.
cy	On exit, overwritten with $-s*x + c*y$.

?laesy

Computes the eigenvalues and eigenvectors of a 2-by-2 complex symmetric matrix, and checks that the norm of the matrix of eigenvectors is larger than a threshold value.

Syntax

```
call claesy( a, b, c, rt1, rt2, evscal, cs1, sn1 )
call zlaesy( a, b, c, rt1, rt2, evscal, cs1, sn1 )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine performs the eigendecomposition of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix},$$

provided the norm of the matrix of eigenvectors is larger than some threshold value.

$rt1$ is the eigenvalue of larger absolute value, and $rt2$ of smaller absolute value. If the eigenvectors are computed, then on return $(cs1, sn1)$ is the unit eigenvector for $rt1$, hence

$$\begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -sn1 \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

Input Parameters

a, b, c	COMPLEX for claesy DOUBLE COMPLEX for zlaesy Elements of the input matrix.
-----------	--

Output Parameters

$rt1, rt2$	COMPLEX for claesy DOUBLE COMPLEX for zlaesy Eigenvalues of larger and smaller modulus, respectively.
$evscal$	COMPLEX for claesy DOUBLE COMPLEX for zlaesy The complex value by which the eigenvector matrix was scaled to make it orthonormal. If $evscal$ is zero, the eigenvectors were not computed. This means one of two things: the 2-by-2 matrix could not be diagonalized, or the norm of the matrix of eigenvectors before scaling was larger than the threshold value $thresh$ (set to 0.1E0).
$cs1, sn1$	COMPLEX for claesy DOUBLE COMPLEX for zlaesy If $evscal$ is not zero, then $(cs1, sn1)$ is the unit right eigenvector for $rt1$.

?rot

Applies a plane rotation with real cosine and complex sine to a pair of complex vectors.

Syntax

```
call crot( n, cx, incx, cy, incy, c, s )
call zrot( n, cx, incx, cy, incy, c, s )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine applies a plane rotation, where the cosine (*c*) is real and the sine (*s*) is complex, and the vectors *cx* and *cy* are complex. This routine has its real equivalents in BLAS (see [?rot](#) in Chapter 2).

Input Parameters

<i>n</i>	INTEGER. The number of elements in the vectors <i>cx</i> and <i>cy</i> .
<i>cx</i> , <i>cy</i>	REAL for srot DOUBLE PRECISION for drot COMPLEX for crot DOUBLE COMPLEX for zrot Arrays of dimension (<i>n</i>), contain input vectors <i>x</i> and <i>y</i> , respectively.
<i>incx</i>	INTEGER. The increment between successive elements of <i>cx</i> .
<i>incy</i>	INTEGER. The increment between successive elements of <i>cy</i> .
<i>c</i>	REAL for crot DOUBLE PRECISION for zrot
<i>s</i>	REAL for srot DOUBLE PRECISION for drot COMPLEX for crot DOUBLE COMPLEX for zrot Values that define a rotation

$$\begin{bmatrix} c & s \\ -\text{conj}(s) & c \end{bmatrix}$$

where $c^*c + s^*\text{conj}(s) = 1.0$.

Output Parameters

<i>cx</i>	On exit, overwritten with $c*x + s*y$.
<i>cy</i>	On exit, overwritten with $-\text{conjg}(s)*x + c*y$.

?spmv

Computes a matrix-vector product for complex vectors using a complex symmetric packed matrix.

Syntax

```
call cspmv( uplo, n, alpha, ap, x, incx, beta, y, incy )
call zspmv( uplo, n, alpha, ap, x, incx, beta, y, incy )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?spmv routines perform a matrix-vector operation defined as

$$y := \alpha * a * x + \beta * y,$$

where:

alpha and *beta* are complex scalars,

x and *y* are *n*-element complex vectors

a is an *n*-by-*n* complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see ?spmv in Chapter 2).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>a</i> is supplied in the packed array <i>ap</i> . If <i>uplo</i> = 'U' or 'u', the upper triangular part of the matrix <i>a</i> is supplied in the array <i>ap</i> . If <i>uplo</i> = 'L' or 'l', the lower triangular part of the matrix <i>a</i> is supplied in the array <i>ap</i> .
<i>n</i>	INTEGER. Specifies the order of the matrix <i>a</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i> , <i>beta</i>	COMPLEX for cspmv DOUBLE COMPLEX for zspmv Specify complex scalars <i>alpha</i> and <i>beta</i> . When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.
<i>ap</i>	COMPLEX for cspmv

DOUBLE COMPLEX for zspmv

Array, DIMENSION at least $((n*(n + 1))/2)$. Before entry, with $uplo = 'U'$ or ' u ', the array ap must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that $ap(1)$ contains $A(1, 1)$, $ap(2)$ and $ap(3)$ contain $A(1, 2)$ and $A(2, 2)$ respectively, and so on. Before entry, with $uplo = 'L'$ or ' l ', the array ap must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that $ap(1)$ contains $a(1, 1)$, $ap(2)$ and $ap(3)$ contain $a(2, 1)$ and $a(3, 1)$ respectively, and so on.

x

COMPLEX for cspmv

DOUBLE COMPLEX for zspmv

Array, DIMENSION at least $(1 + (n - 1)*abs(incx))$. Before entry, the incremented array x must contain the n -element vector x .

incx

INTEGER. Specifies the increment for the elements of x . The value of $incx$ must not be zero.

y

COMPLEX for cspmv

DOUBLE COMPLEX for zspmv

Array, DIMENSION at least $(1 + (n - 1)*abs(incy))$. Before entry, the incremented array y must contain the n -element vector y .

incy

INTEGER. Specifies the increment for the elements of y . The value of $incy$ must not be zero.

Output Parameters

y

Overwritten by the updated vector y .

?spr

Performs the symmetrical rank-1 update of a complex symmetric packed matrix.

Syntax

```
call cspr( uplo, n, alpha, x, incx, ap )
call zspr( uplo, n, alpha, x, incx, ap )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?spr routines perform a matrix-vector operation defined as

$$a := \alpha * x * x^H + a,$$

where:

α is a complex scalar

x is an n -element complex vector

a is an n -by- n complex symmetric matrix, supplied in packed form.

These routines have their real equivalents in BLAS (see [?spr](#) in Chapter 2).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the matrix <i>a</i> is supplied in the packed array <i>ap</i> , as follows: If <i>uplo</i> = 'U' or 'u', the upper triangular part of the matrix <i>a</i> is supplied in the array <i>ap</i> . If <i>uplo</i> = 'L' or 'l', the lower triangular part of the matrix <i>a</i> is supplied in the array <i>ap</i> .
<i>n</i>	INTEGER. Specifies the order of the matrix <i>a</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	COMPLEX for cspr DOUBLE COMPLEX for zspr Specifies the scalar <i>alpha</i> .
<i>x</i>	COMPLEX for cspr DOUBLE COMPLEX for zspr Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the n -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>ap</i>	COMPLEX for cspr DOUBLE COMPLEX for zspr Array, DIMENSION at least $((n*(n + 1)) / 2)$. Before entry, with <i>uplo</i> = 'U' or 'u', the array <i>ap</i> must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains $A(1, 1)$, <i>ap</i> (2) and <i>ap</i> (3) contain $A(1, 2)$ and $A(2, 2)$ respectively, and so on. Before entry, with <i>uplo</i> = 'L' or 'l', the array <i>ap</i> must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that <i>ap</i> (1) contains $A(1, 1)$, <i>ap</i> (2) and <i>ap</i> (3) contain $A(2, 1)$ and $A(3, 1)$ respectively, and so on. Note that the imaginary parts of the diagonal elements need not be set, they are assumed to be zero, and on exit they are set to zero.

Output Parameters

<i>ap</i>	With <i>uplo</i> = 'U' or 'u', overwritten by the upper triangular part of the updated matrix.
-----------	--

With $uplo = 'L'$ or $'U'$, overwritten by the lower triangular part of the updated matrix.

?syconv

Converts a symmetric matrix given by a triangular matrix factorization into two matrices and vice versa.

Syntax

FORTRAN 77:

```
call ssyconv( uplo, way, n, a, lda, ipiv, work, info )
call dsyconv( uplo, way, n, a, lda, ipiv, work, info )
call csyconv( uplo, way, n, a, lda, ipiv, work, info )
call zsyconv( uplo, way, n, a, lda, ipiv, work, info )
```

Fortran 95:

```
call syconv( a[,uplo][,way][,ipiv][,info] )
```

C:

```
lapack_int LAPACKE_<?>syconv (int matrix_layout, char uplo, char way, lapack_int n,
<datatype> * a, lapack_int lda, const lapack_int * ipiv);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine converts matrix A , which results from a triangular matrix factorization, into matrices L and D and vice versa. The routine gets non-diagonalized elements of D returned in the workspace and applies or reverses permutation done with the triangular matrix factorization.

Input Parameters

The data types are given for the Fortran interface.

<i>uplo</i>	CHARACTER*1. Must be ' U ' or ' L '.
	Indicates whether the details of the factorization are stored as an upper or lower triangular matrix: If $uplo = 'U'$: the upper triangular, $A = U*D*U^T$. If $uplo = 'L'$: the lower triangular, $A = L*D*L^T$.
<i>way</i>	CHARACTER*1. Must be ' C ' or ' R '.
	Indicates whether the routine converts or reverts the matrix: $way = 'C'$ means conversion. $way = 'R'$ means reversion.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.

<i>a</i>	REAL for ssyconv DOUBLE PRECISION for dsyconv COMPLEX for csyconv DOUBLE COMPLEX for zsyconv Array of DIMENSION (<i>lda,n</i>). The block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> as computed by ?sytrf.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; <i>lda</i> $\geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. Details of the interchanges and the block structure of <i>D</i> , as returned by ?sytrf.
<i>work</i>	INTEGER. Workspace array, DIMENSION at least $\max(1, n)$.

Output Parameters

<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, the <i>i</i> -th parameter had an illegal value.
-------------	---

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `syconv` interface are as follows:

<i>a</i>	Holds the matrix <i>A</i> of size (<i>n, n</i>).
<i>uplo</i>	Must be 'U' or 'L'.
<i>way</i>	Must be 'C' or 'R'.
<i>ipiv</i>	Holds the vector of length <i>n</i> .

See Also

[?sytrf](#)

?symv

Computes a matrix-vector product for a complex symmetric matrix.

Syntax

```
call csymv( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
call zsymv( uplo, n, alpha, a, lda, x, incx, beta, y, incy )
```

Include Files

- Fortran: `mkl.fi`

-
- C: mkl.h

Description

The routine performs the matrix-vector operation defined as

$$y := \alpha * a * x + \beta * y,$$

where:

alpha and *beta* are complex scalars

x and *y* are *n*-element complex vectors

a is an *n*-by-*n* symmetric complex matrix.

These routines have their real equivalents in BLAS (see [?symv](#) in Chapter 2).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>a</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>a</i> is used. If <i>uplo</i> = 'L' or 'l', then the lower triangular part of the array <i>a</i> is used.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>a</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i> , <i>beta</i>	COMPLEX for <code>csymv</code> DOUBLE COMPLEX for <code>zsymv</code> Specify the scalars <i>alpha</i> and <i>beta</i> . When <i>beta</i> is supplied as zero, then <i>y</i> need not be set on input.
<i>a</i>	COMPLEX for <code>csymv</code> DOUBLE COMPLEX for <code>zsymv</code> Array, DIMENSION (<i>lda</i> , <i>n</i>). Before entry with <i>uplo</i> = 'U' or 'u', the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of <i>a</i> is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', the leading <i>n</i> -by- <i>n</i> lower triangular part of the array <i>a</i> must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>A</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least <code>max(1, n)</code> .
<i>x</i>	COMPLEX for <code>csymv</code> DOUBLE COMPLEX for <code>zsymv</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array <i>x</i> must contain the <i>n</i> -element vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.
<i>y</i>	COMPLEX for <code>csymv</code>

DOUBLE COMPLEX for zsymv

Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incy))$. Before entry, the incremented array y must contain the n -element vector y .

incy INTEGER. Specifies the increment for the elements of y . The value of *incy* must not be zero.

Output Parameters

y Overwritten by the updated vector y .

?syr

Performs the symmetric rank-1 update of a complex symmetric matrix.

Syntax

FORTRAN 77:

```
call csyr( uplo, n, alpha, x, incx, a, lda )
call zsyr( uplo, n, alpha, x, incx, a, lda )
```

C:

```
lapack_int LAPACKE_csy (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float alpha, const lapack_complex_float * x, lapack_int incx,
lapack_complex_float * a, lapack_int lda);

lapack_int LAPACKE_zsy (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double alpha, const lapack_complex_double * x, lapack_int incx,
lapack_complex_double * a, lapack_int lda);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine performs the symmetric rank 1 operation defined as

$$a := \alpha * x * x^H + a,$$

where:

- α is a complex scalar.
- x is an n -element complex vector.
- a is an n -by- n complex symmetric matrix.

These routines have their real equivalents in BLAS (see [?syr](#) in Chapter 2).

Input Parameters

uplo CHARACTER*1. Specifies whether the upper or lower triangular part of the array a is used:
If $uplo = 'U'$ or 'u', then the upper triangular part of the array a is used.

If $uplo = 'L'$ or $'l'$, then the lower triangular part of the array a is used.

n INTEGER. Specifies the order of the matrix a . The value of n must be at least zero.

α COMPLEX for csyr

DOUBLE COMPLEX for zsy r

Specifies the scalar α .

x COMPLEX for csyr

DOUBLE COMPLEX for zsy r

Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the n -element vector x .

$incx$ INTEGER. Specifies the increment for the elements of x . The value of $incx$ must not be zero.

a COMPLEX for csyr

DOUBLE COMPLEX for zsy r

Array, DIMENSION (lda, n). Before entry with $uplo = 'U'$ or $'u'$, the leading n -by- n upper triangular part of the array a must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of a is not referenced.

Before entry with $uplo = 'L'$ or $'l'$, the leading n -by- n lower triangular part of the array a must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of a is not referenced.

lda INTEGER. Specifies the leading dimension of a as declared in the calling (sub)program. The value of lda must be at least $\max(1, n)$.

Output Parameters

a With $uplo = 'U'$ or $'u'$, the upper triangular part of the array a is overwritten by the upper triangular part of the updated matrix.

With $uplo = 'L'$ or $'l'$, the lower triangular part of the array a is overwritten by the lower triangular part of the updated matrix.

i?max1

Finds the index of the vector element whose real part has maximum absolute value.

Syntax

$index = \text{icmax1}(n, cx, incx)$

$index = \text{izmax1}(n, cx, incx)$

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

Given a complex vector \mathbf{cx} , the $i?max1$ functions return the index of the vector element whose real part has maximum absolute value. These functions are based on the BLAS functions $icamax/izamax$, but using the absolute value of the real part. They are designed for use with $clacon/zlacon$.

Input Parameters

n	INTEGER. Specifies the number of elements in the vector \mathbf{cx} .
\mathbf{cx}	COMPLEX for $icmax1$ DOUBLE COMPLEX for $izmax1$ Array, DIMENSION at least $(1+(n-1)*abs(incx))$. Contains the input vector.
$incx$	INTEGER. Specifies the spacing between successive elements of \mathbf{cx} .

Output Parameters

$index$	INTEGER. Contains the index of the vector element whose real part has maximum absolute value.
---------	---

?sum1

Forms the 1-norm of the complex vector using the true absolute value.

Syntax

```
res = scsum1( n, cx, incx )
res = dzsum1( n, cx, incx )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

Given a complex vector \mathbf{cx} , $scsum1/dzsum1$ functions take the sum of the absolute values of vector elements and return a single/double precision result, respectively. These functions are based on $scasum/dzasum$ from Level 1 BLAS, but use the true absolute value and were designed for use with $clacon/zlacon$.

Input Parameters

n	INTEGER. Specifies the number of elements in the vector \mathbf{cx} .
\mathbf{cx}	COMPLEX for $scsum1$ DOUBLE COMPLEX for $dzsum1$ Array, DIMENSION at least $(1+(n-1)*abs(incx))$. Contains the input vector whose elements will be summed.

incx INTEGER. Specifies the spacing between successive elements of *cx* (*incx* > 0).

Output Parameters

<i>res</i>	REAL for <i>sctsum1</i>
	DOUBLE PRECISION for <i>dzsum1</i>
	Contains the sum of absolute values.

?gbtf2

Computes the LU factorization of a general band matrix using the unblocked version of the algorithm.

Syntax

```
call sgbt2( m, n, kl, ku, ab, ldab, ipiv, info )
call dgbt2( m, n, kl, ku, ab, ldab, ipiv, info )
call cgbt2( m, n, kl, ku, ab, ldab, ipiv, info )
call zgbt2( m, n, kl, ku, ab, ldab, ipiv, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine forms the *LU* factorization of a general real/complex *m*-by-*n* band matrix *A* with *kl* sub-diagonals and *ku* super-diagonals. The routine uses partial pivoting with row interchanges and implements the unblocked version of the algorithm, calling Level 2 BLAS. See also [?gbtrf](#).

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> (<i>m</i> ≥ 0).
<i>n</i>	INTEGER. The number of columns in <i>A</i> (<i>n</i> ≥ 0).
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of <i>A</i> (<i>kl</i> ≥ 0).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of <i>A</i> (<i>ku</i> ≥ 0).
<i>ab</i>	REAL for sgbt2 DOUBLE PRECISION for dgbt2 COMPLEX for cgbt2 DOUBLE COMPLEX for zgbt2. Array, DIMENSION (ldab,*). The array <i>ab</i> contains the matrix <i>A</i> in band storage (see Matrix Arguments). The second dimension of <i>ab</i> must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of the array *ab*.
 $(ldab \geq 2kl + ku + 1)$

Output Parameters

ab Overwritten by details of the factorization. The diagonal and $kl + ku$ super-diagonals of *U* are stored in the first $1 + kl + ku$ rows of *ab*. The multipliers used during the factorization are stored in the next *kl* rows.

ipiv INTEGER.
Array, DIMENSION at least $\max(1, \min(m, n))$.
The pivot indices: row *i* was interchanged with row *ipiv(i)*.

info INTEGER. If *info* = 0, the execution is successful.
If *info* = *-i*, the *i*-th parameter had an illegal value.
If *info* = *i*, u_{ii} is 0. The factorization has been completed, but *U* is exactly singular. Division by 0 will occur if you use the factor *U* for solving a system of linear equations.

?gebd2

Reduces a general matrix to bidiagonal form using an unblocked algorithm.

Syntax

```
call sgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call dgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call cgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
call zgebd2( m, n, a, lda, d, e, tauq, taup, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine reduces a general *m*-by-*n* matrix *A* to upper or lower bidiagonal form *B* by an orthogonal (unitary) transformation: $Q^T A P = B$ (for real flavors) or $Q^H A P = B$ (for complex flavors).

If *m* \geq *n*, *B* is upper bidiagonal; if *m* < *n*, *B* is lower bidiagonal.

The routine does not form the matrices *Q* and *P* explicitly, but represents them as products of elementary reflectors. If *m* \geq *n*,

$Q = H(1) * H(2) * \dots * H(n)$, and $P = G(1) * G(2) * \dots * G(n-1)$

if *m* < *n*,

$Q = H(1) * H(2) * \dots * H(m-1)$, and $P = G(1) * G(2) * \dots * G(m)$

Each *H(i)* and *G(i)* has the form

$H(i) = I - tauq * v * v^T$ and $G(i) = I - taup * u * u^T$ for real flavors, or

$H(i) = I - \tau_{aq} v^* v^H$ and $G(i) = I - \tau_{ap} u^* u^H$ for complex flavors

where τ_{aq} and τ_{ap} are scalars (real for `sgebd2/dgebd2`, complex for `cgebd2/zgebd2`), and v and u are vectors (real for `sgebd2/dgebd2`, complex for `cgebd2/zgebd2`).

Input Parameters

`m` INTEGER. The number of rows in the matrix A ($m \geq 0$).

`n` INTEGER. The number of columns in A ($n \geq 0$).

`a, work` REAL for `sgebd2`

DOUBLE PRECISION for `dgebd2`

COMPLEX for `cgebd2`

DOUBLE COMPLEX for `zgebd2`.

Arrays:

`a(1da, *)` contains the m -by- n general matrix A to be reduced. The second dimension of `a` must be at least $\max(1, n)$.

`work(*)` is a workspace array, the dimension of `work` must be at least $\max(1, m, n)$.

`lda` INTEGER. The leading dimension of `a`; at least $\max(1, m)$.

Output Parameters

`a` if $m \geq n$, the diagonal and first super-diagonal of `a` are overwritten with the upper bidiagonal matrix B . Elements below the diagonal, with the array `tauq`, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements above the first superdiagonal, with the array `taup`, represent the orthogonal/unitary matrix P as a product of elementary reflectors.

if $m < n$, the diagonal and first sub-diagonal of `a` are overwritten by the lower bidiagonal matrix B . Elements below the first subdiagonal, with the array `tauq`, represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements above the diagonal, with the array `taup`, represent the orthogonal/unitary matrix P as a product of elementary reflectors.

`d` REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

Array, DIMENSION at least $\max(1, \min(m, n))$.

Contains the diagonal elements of the bidiagonal matrix B : $d(i) = a(i, i)$.

`e` REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors. Array, DIMENSION at least $\max(1, \min(m, n) - 1)$.

Contains the off-diagonal elements of the bidiagonal matrix B :

if $m \geq n$, $e(i) = a(i, i+1)$ for $i = 1, 2, \dots, n-1$;

```

if  $m < n$ ,  $e(i) = a(i+1, i)$  for  $i = 1, 2, \dots, m-1$ .
taug, taup          REAL for sgebd2
                    DOUBLE PRECISION for dgebd2
                    COMPLEX for cgebd2
                    DOUBLE COMPLEX for zgebd2.

Arrays, DIMENSION at least  $\max(1, \min(m, n))$ .
Contain scalar factors of the elementary reflectors which represent
orthogonal/unitary matrices  $Q$  and  $p$ , respectively.

info               INTEGER.
If  $info = 0$ , the execution is successful.
If  $info = -i$ , the  $i$ th parameter had an illegal value.

```

?gehd2

Reduces a general square matrix to upper Hessenberg form using an unblocked algorithm.

Syntax

```

call sgehd2(  $n, ilo, ihi, a, lda, tau, work, info$  )
call dgehd2(  $n, ilo, ihi, a, lda, tau, work, info$  )
call cgehd2(  $n, ilo, ihi, a, lda, tau, work, info$  )
call zgehd2(  $n, ilo, ihi, a, lda, tau, work, info$  )

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine reduces a real/complex general matrix A to upper Hessenberg form H by an orthogonal or unitary similarity transformation $Q^T A Q = H$ (for real flavors) or $Q^H A Q = H$ (for complex flavors).

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of *elementary reflectors*.

Input Parameters

n	INTEGER The order of the matrix A ($n \geq 0$).
ilo, ihi	INTEGER. It is assumed that A is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$. If A has been output by ?gebal, then ilo and ihi must contain the values returned by that routine. Otherwise they should be set to $ilo = 1$ and $ihi = n$. Constraint: $1 \leq ilo \leq ihi \leq \max(1, n)$.

a, work

REAL for sgehd2
DOUBLE PRECISION for dgehd2
COMPLEX for cgehd2
DOUBLE COMPLEX for zgehd2.

Arrays:

a (*lda, **) contains the n -by- n matrix A to be reduced. The second dimension of *a* must be at least $\max(1, n)$.
work (*n*) is a workspace array.

lda

INTEGER. The leading dimension of *a*; at least $\max(1, n)$.

Output Parameters

a

On exit, the upper triangle and the first subdiagonal of A are overwritten with the upper Hessenberg matrix H and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors. See *Application Notes* below.

tau

REAL for sgehd2
DOUBLE PRECISION for dgehd2
COMPLEX for cgehd2
DOUBLE COMPLEX for zgehd2.

Array, DIMENSION at least $\max(1, n-1)$.
Contains the scalar factors of elementary reflectors. See *Application Notes* below.

info

INTEGER.
If *info* = 0, the execution is successful.
If *info* = $-i$, the *i*-th parameter had an illegal value.

Application Notes

The matrix Q is represented as a product of (*ih* - *ilo*) elementary reflectors

$$Q = H(i_0) * H(i_0 + 1) * \dots * H(i_h - 1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v^T \text{ for real flavors, or}$$

$$H(i) = I - \tau * v * v^H \text{ for complex flavors}$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$, $v(i+1) = 1$ and $v(i_h + 1:n) = 0$.

On exit, $v(i+2:i_h)$ is stored in *a*(*i*+2:*i_h*, *i*) and τ in *tau*(*i*).

The contents of *a* are illustrated by the following example, with $n = 7$, *ilo* = 2 and *ih* = 6:

on entry	on exit
$\begin{bmatrix} a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \end{bmatrix}$	$\begin{bmatrix} a & a & h & h & h & h & a \\ a & h & h & h & h & h & a \\ h & h & h & h & h & h & h \\ v_2 & h & h & h & h & h & h \\ v_2 & v_3 & h & h & h & h & h \\ v_2 & v_3 & v_4 & h & h & h & h \\ a & & & & & & a \end{bmatrix}$

where a denotes an element of the original matrix A , h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(i)$.

?gelq2

Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgelq2( m, n, a, lda, tau, work, info )
call dgelq2( m, n, a, lda, tau, work, info )
call cgelq2( m, n, a, lda, tau, work, info )
call zgelq2( m, n, a, lda, tau, work, info )
```

C:

```
lapack_int LAPACKE_<?>gelq2 (int matrix_layout, lapack_int m, lapack_int n, <datatype>
* a, lapack_int lda, <datatype> * tau);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes an LQ factorization of a real/complex m -by- n matrix A as $A = L^*Q$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors :

$Q = H(k) \dots H(2) H(1)$ (or $Q = H(k)^H \dots H(2)^H H(1)^H$ for complex flavors), where $k = \min(m, n)$

Each $H(i)$ has the form

$H(i) = I - tau * v * v^T$ for real flavors, or

$H(i) = I - tau * v * v^H$ for complex flavors,

where tau is a real/complex scalar stored in $tau(i)$, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$.

On exit, $v(i+1:n)$ (for real functions) and $\text{conjgv}(i+1:n)$ (for complex functions) are stored in $a(i, i+1:n)$.

Input Parameters

The data types are given for the Fortran interface.

m	INTEGER. The number of rows in the matrix A ($m \geq 0$).
n	INTEGER. The number of columns in A ($n \geq 0$).
$a, work$	REAL for <code>sgeql2</code> DOUBLE PRECISION for <code>dgeql2</code> COMPLEX for <code>cgeql2</code> DOUBLE COMPLEX for <code>zgeql2</code> . Arrays: $a(lda,*)$ contains the m -by- n matrix A . The second dimension of a must be at least $\max(1, n)$. $work(m)$ is a workspace array.
lda	INTEGER. The leading dimension of a ; at least $\max(1, m)$.

Output Parameters

a	Overwritten by the factorization data as follows: on exit, the elements on and below the diagonal of the array a contain the m -by- $\min(n,m)$ lower trapezoidal matrix L (L is lower triangular if $n \geq m$); the elements above the diagonal, with the array tau , represent the orthogonal/unitary matrix Q as a product of $\min(n,m)$ elementary reflectors.
tau	REAL for <code>sgeql2</code> DOUBLE PRECISION for <code>dgeql2</code> COMPLEX for <code>cgeql2</code> DOUBLE COMPLEX for <code>zgeql2</code> . Array, DIMENSION at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

?geql2

Computes the QL factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeql2( m, n, a, lda, tau, work, info )
call dgeql2( m, n, a, lda, tau, work, info )
```

```
call cgeql2( m, n, a, lda, tau, work, info )
call zgeql2( m, n, a, lda, tau, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes a QL factorization of a real/complex m -by- n matrix A as $A = Q^* L$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors :

$$Q = H(k) * \dots * H(2) * H(1), \text{ where } k = \min(m, n).$$

Each $H(i)$ has the form

$$H(i) = I - tau * v * v^T \text{ for real flavors, or}$$

$$H(i) = I - tau * v * v^H \text{ for complex flavors}$$

where tau is a real/complex scalar stored in $tau(i)$, and v is a real/complex vector with $v(m-k+i+1:m) = 0$ and $v(m-k+i) = 1$.

On exit, $v(1:m-k+i-1)$ is stored in $a(1:m-k+i-1, n-k+i)$.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a, work REAL for sgeql2

DOUBLE PRECISION for dgeql2

COMPLEX for cgeql2

DOUBLE COMPLEX for zgeql2.

Arrays:

a(*lda*,*) contains the m -by- n matrix A .

The second dimension of *a* must be at least $\max(1, n)$.

work(*m*) is a workspace array.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

Output Parameters

a

Overwritten by the factorization data as follows:

on exit, if $m \geq n$, the lower triangle of the subarray $a(m-n+1:m, 1:n)$ contains the n -by- n lower triangular matrix L ; if $m < n$, the elements on and below the $(n-m)$ th superdiagonal contain the m -by- n lower trapezoidal matrix L ; the remaining elements, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

tau

REAL for sgeql2

DOUBLE PRECISION for dgeql2
 COMPLEX for cgeql2
 DOUBLE COMPLEX for zgeql2.
Array, DIMENSION at least $\max(1, \min(m, n))$.
 Contains scalar factors of the elementary reflectors.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value.

?geqr2

Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgeqr2( m, n, a, lda, tau, work, info )
call dgeqr2( m, n, a, lda, tau, work, info )
call cgeqr2( m, n, a, lda, tau, work, info )
call zgeqr2( m, n, a, lda, tau, work, info )
```

C:

```
lapack_int LAPACKE_<?>geqr2 (int matrix_layout, lapack_int m, lapack_int n, <datatype>
* a, lapack_int lda, <datatype> * tau);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes a QR factorization of a real/complex m -by- n matrix A as $A = Q^*R$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors :

$Q = H(1)^*H(2)^* \dots ^*H(k)$, where $k = \min(m, n)$

Each $H(i)$ has the form

$H(i) = I - tau*v*v^T$ for real flavors, or

$H(i) = I - tau*v*v^H$ for complex flavors

where tau is a real/complex scalar stored in $tau(i)$, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$.

On exit, $v(i+1:m)$ is stored in $a(i+1:m, i)$.

Input Parameters

The data types are given for the Fortran interface.

<i>m</i>	INTEGER. The number of rows in the matrix <i>A</i> ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in <i>A</i> ($n \geq 0$).
<i>a</i> , <i>work</i>	REAL for sgeqr2 DOUBLE PRECISION for dgeqr2 COMPLEX for cgeqr2 DOUBLE COMPLEX for zgeqr2.
	Arrays: <i>a</i> (<i>lda</i> , *) contains the <i>m</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>work</i> (<i>n</i>) is a workspace array.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; at least $\max(1, m)$.

Output Parameters

<i>a</i>	Overwritten by the factorization data as follows: on exit, the elements on and above the diagonal of the array <i>a</i> contain the $\min(n,m)$ -by- <i>n</i> upper trapezoidal matrix <i>R</i> (<i>R</i> is upper triangular if $m \geq n$); the elements below the diagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of elementary reflectors.
<i>tau</i>	REAL for sgeqr2 DOUBLE PRECISION for dgeqr2 COMPLEX for cgeqr2 DOUBLE COMPLEX for zgeqr2. Array, DIMENSION at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

?geqr2p

Computes the QR factorization of a general rectangular matrix with non-negative diagonal elements using an unblocked algorithm.

Syntax

```
call sgeqr2p( m, n, a, lda, tau, work, info )
call dgeqr2p( m, n, a, lda, tau, work, info )
call cgeqr2p( m, n, a, lda, tau, work, info )
call zgeqr2p( m, n, a, lda, tau, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes a QR factorization of a real/complex m -by- n matrix A as $A = Q^*R$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors :

$$Q = H(1)^*H(2)^* \dots ^*H(k), \text{ where } k = \min(m, n)$$

Each $H(i)$ has the form

$$H(i) = I - tau * v * v^T \text{ for real flavors, or}$$

$$H(i) = I - tau * v * v^H \text{ for complex flavors}$$

where tau is a real/complex scalar stored in $tau(i)$, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$.

On exit, $v(i+1:m)$ is stored in $a(i+1:m, i)$.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

$a, work$ REAL for sgeqr2p

DOUBLE PRECISION for d

COMPLEX for cgeqr2p

DOUBLE COMPLEX for zgeqr2p.

Arrays:

$a(lda, *)$ contains the m -by- n matrix A .

The second dimension of a must be at least $\max(1, n)$.

$work(n)$ is a workspace array.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

Output Parameters

a Overwritten by the factorization data as follows:

on exit, the elements on and above the diagonal of the array a contain the $\min(n,m)$ -by- n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

The diagonal elements of the matrix R are non-negative.

tau REAL for sgeqr2p

DOUBLE PRECISION for dgeqr2p

COMPLEX for cgeqr2p

DOUBLE COMPLEX for zgeqr2p.

Array, DIMENSION at least $\max(1, \min(m, n))$.

Contains scalar factors of the elementary reflectors.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

?geqrt2

Computes a QR factorization of a general real or complex matrix using the compact WY representation of Q .

Syntax

FORTRAN 77:

```
call sgeqrt2(m, n, a, lda, t, ldt, info)
call dgeqrt2(m, n, a, lda, t, ldt, info)
call cgeqrt2(m, n, a, lda, t, ldt, info)
call zgeqrt2(m, n, a, lda, t, ldt, info)
```

Fortran 95:

```
call geqrt2(a, t, [info])
```

C:

```
lapack_int LAPACKE_sgeqrt2 (int matrix_layout, lapack_int m, lapack_int n, float * a,
lapack_int lda, float * t, lapack_int ldt );
lapack_int LAPACKE_dgeqrt2 (int matrix_layout, lapack_int m, lapack_int n, double * a,
lapack_int lda, double * t, lapack_int ldt );
lapack_int LAPACKE_cgeqrt2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float * a, lapack_int lda, lapack_complex_float * t, lapack_int ldt );
lapack_int LAPACKE_zgeqrt2 (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double * a, lapack_int lda, lapack_complex_double * t, lapack_int ldt );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The strictly lower triangular matrix V contains the elementary reflectors $H(i)$ in the i th column below the diagonal. For example, if $m=5$ and $n=3$, the matrix V is

$$V = \begin{bmatrix} 1 & & \\ v_1 & 1 & \\ v_1 & v_2 & 1 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

where v_i represents one of the vectors that define $H(i)$. The vectors are returned in the lower triangular part of array a .

NOTE

The 1s along the diagonal of V are not stored in a .

The block reflector H is then given by

$H = I - V^* T^* V^T$ for real flavors, and

$H = I - V^* T^* V^H$ for complex flavors,

where V^T is the transpose and V^H is the conjugate transpose of V .

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq n$).

n INTEGER. The number of columns in A ($n \geq 0$).

a REAL for sgeqrt2

DOUBLE PRECISION for dgeqrt2

COMPLEX for cgeqrt2

COMPLEX*16 for zgeqrt2.

Arrays: *a* DIMENSION (*lda*, *n*) contains the m -by- n matrix A .

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

ldt INTEGER. The leading dimension of *t*; at least $\max(1, n)$.

Output Parameters

a Overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the n -by- n upper triangular matrix R . The elements below the diagonal are the columns of V .

t REAL for sgeqrt2

DOUBLE PRECISION for dgeqrt2

COMPLEX for cgeqrt2

COMPLEX*16 for zgeqrt2.

`Array, DIMENSION (l_{dt} , $\min(m, n)$).`

The n -by- n upper triangular factor of the block reflector. The elements on and above the diagonal contain the block reflector T . The elements below the diagonal are not used.

`info`

INTEGER.

If `info` = 0, the execution is successful.

If `info` < 0 and `info` = $-i$, the i th argument had an illegal value.

?geqrt3

Recursively computes a QR factorization of a general real or complex matrix using the compact WY representation of Q .

Syntax

FORTRAN 77:

```
call sgeqrt3( $m$ ,  $n$ ,  $a$ ,  $lda$ ,  $t$ ,  $l_{dt}$ , info)
call dgeqrt3( $m$ ,  $n$ ,  $a$ ,  $lda$ ,  $t$ ,  $l_{dt}$ , info)
call cgeqrt3( $m$ ,  $n$ ,  $a$ ,  $lda$ ,  $t$ ,  $l_{dt}$ , info)
call zgeqrt3( $m$ ,  $n$ ,  $a$ ,  $lda$ ,  $t$ ,  $l_{dt}$ , info)
```

Fortran 95:

```
call geqrt3( $a$ ,  $t$  [, info])
```

C:

```
lapack_int LAPACKE_sgeqrt3 (int matrix_layout , lapack_int  $m$  , lapack_int  $n$  , float *  $a$  , lapack_int  $lda$  , float *  $t$  , lapack_int  $l_{dt}$  );
lapack_int LAPACKE_dgeqrt3 (int matrix_layout , lapack_int  $m$  , lapack_int  $n$  , double *  $a$  , lapack_int  $lda$  , double *  $t$  , lapack_int  $l_{dt}$  );
lapack_int LAPACKE_cgeqrt3 (int matrix_layout , lapack_int  $m$  , lapack_int  $n$  ,
lapack_complex_float *  $a$  , lapack_int  $lda$  , lapack_complex_float *  $t$  , lapack_int  $l_{dt}$  );
lapack_int LAPACKE_zgeqrt3 (int matrix_layout , lapack_int  $m$  , lapack_int  $n$  ,
lapack_complex_double *  $a$  , lapack_int  $lda$  , lapack_complex_double *  $t$  , lapack_int  $l_{dt}$  );
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The strictly lower triangular matrix V contains the elementary reflectors $H(i)$ in the i th column below the diagonal. For example, if $m=5$ and $n=3$, the matrix V is

$$V = \begin{bmatrix} 1 & & \\ v_1 & 1 & \\ v_1 & v_2 & 1 \\ v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

where v_i represents one of the vectors that define $H(i)$. The vectors are returned in the lower part of triangular array a .

NOTE

The 1s along the diagonal of V are not stored in a .

The block reflector H is then given by

$H = I - V^T T^* V^T$ for real flavors, and

$H = I - V^H T^* V^H$ for complex flavors,

where V^T is the transpose and V^H is the conjugate transpose of V .

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq n$).

n INTEGER. The number of columns in A ($n \geq 0$).

a REAL for sgeqrt3

DOUBLE PRECISION for dgeqrt3

COMPLEX for cgeqrt3

COMPLEX*16 for zgeqrt3.

Arrays: *a* DIMENSION (*lda*, *n*) contains the m -by- n matrix A .

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

ldt INTEGER. The leading dimension of *t*; at least $\max(1, n)$.

Output Parameters

a The elements on and above the diagonal of the array contain the n -by- n upper triangular matrix R . The elements below the diagonal are the columns of V .

t REAL for sgeqrt3

DOUBLE PRECISION for dgeqrt3

COMPLEX for cgeqrt3

COMPLEX*16 for zgeqrt3.

Array, DIMENSION (*ldt*, *n*).

The n -by- n upper triangular factor of the block reflector. The elements on and above the diagonal contain the block reflector T . The elements below the diagonal are not used.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0 and *info* = $-i$, the *i*th argument had an illegal value.

?gerq2

Computes the RQ factorization of a general rectangular matrix using an unblocked algorithm.

Syntax

```
call sgerq2( m, n, a, lda, tau, work, info )
call dgerq2( m, n, a, lda, tau, work, info )
call cgerq2( m, n, a, lda, tau, work, info )
call zgerq2( m, n, a, lda, tau, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes a *RQ* factorization of a real/complex m -by- n matrix A as $A = R*Q$.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of $\min(m, n)$ elementary reflectors :

$Q = H(1) * H(2) * \dots * H(k)$ for real flavors, or

$Q = H(1)^H * H(2)^H * \dots * H(k)^H$ for complex flavors

where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - tau * v * v^T$ for real flavors, or

$H(i) = I - tau * v * v^H$ for complex flavors

where tau is a real/complex scalar stored in $tau(i)$, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$.

On exit, $v(1:n-k+i-1)$ is stored in $a(m-k+i, 1:n-k+i-1)$.

Input Parameters

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a, work REAL for sgerq2

DOUBLE PRECISION for dgerq2

COMPLEX for `cgerq2`
 DOUBLE COMPLEX for `zgerq2`.

Arrays:

`a(1da,*)` contains the m -by- n matrix A .
 The second dimension of `a` must be at least `max(1, n)`.
`work(m)` is a workspace array.

`lda` INTEGER. The leading dimension of `a`; at least `max(1, m)`.

Output Parameters

`a` Overwritten by the factorization data as follows:
 on exit, if $m \leq n$, the upper triangle of the subarray `a(1:m, n-m+1:n)` contains the m -by- m upper triangular matrix R ; if $m > n$, the elements on and above the $(m-n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array `tau`, represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

`tau` REAL for `sgerq2`
 DOUBLE PRECISION for `dgerq2`
 COMPLEX for `cgerq2`
 DOUBLE COMPLEX for `zgerq2`.
Array, DIMENSION at least `max(1, min(m, n))`.
 Contains scalar factors of the elementary reflectors.

`info` INTEGER.
 If `info = 0`, the execution is successful.
 If `info = -i`, the i -th parameter had an illegal value.

?gesc2

Solves a system of linear equations using the LU factorization with complete pivoting computed by ?getc2.

Syntax

```
call sgesc2( n, a, lda, rhs, ipiv, jpivot, scale )
call dgesc2( n, a, lda, rhs, ipiv, jpivot, scale )
call cgesc2( n, a, lda, rhs, ipiv, jpivot, scale )
call zgesc2( n, a, lda, rhs, ipiv, jpivot, scale )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine solves a system of linear equations

$$A \cdot X = scale \cdot RHS$$

with a general n -by- n matrix A using the LU factorization with complete pivoting computed by [?getc2](#).

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A .
<i>a, rhs</i>	REAL for sgesc2 DOUBLE PRECISION for dgesc2 COMPLEX for cgesc2 DOUBLE COMPLEX for zgesc2. Arrays: $a(1:da, *)$ contains the LU part of the factorization of the n -by- n matrix A computed by ?getc2 : $A = P \cdot L \cdot U \cdot Q.$ The second dimension of a must be at least $\max(1, n)$; $rhs(n)$ contains on entry the right hand side vector for the system of equations.
<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The pivot indices: for $1 \leq i \leq n$, row i of the matrix has been interchanged with row $ipiv(i)$.
<i>jpivot</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The pivot indices: for $1 \leq j \leq n$, column j of the matrix has been interchanged with column $jpivot(j)$.

Output Parameters

<i>rhs</i>	On exit, overwritten with the solution vector X .
<i>scale</i>	REAL for sgesc2/cgesc2 DOUBLE PRECISION for dgesc2/zgesc2 Contains the scale factor. <i>scale</i> is chosen in the range $0 \leq scale \leq 1$ to prevent overflow in the solution.

?getc2

Computes the LU factorization with complete pivoting of the general n -by- n matrix.

Syntax

```
call sgetc2( n, a, lda, ipiv, jpivot, info )
call dgetc2( n, a, lda, ipiv, jpivot, info )
call cgetc2( n, a, lda, ipiv, jpivot, info )
call zgetc2( n, a, lda, ipiv, jpivot, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes an *LU* factorization with complete pivoting of the n -by- n matrix A . The factorization has the form $A = P^* L^* U^* Q$, where P and Q are permutation matrices, L is lower triangular with unit diagonal elements and U is upper triangular.

The LU factorization computed by this routine is used by [?latdf](#) to compute a contribution to the reciprocal Dif-estimate.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>a</i>	REAL for sgetc2 DOUBLE PRECISION for dgetc2 COMPLEX for cgetc2 DOUBLE COMPLEX for zgetc2. Array $a(lda, *)$ contains the n -by- n matrix A to be factored. The second dimension of a must be at least $\max(1, n)$;
<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, n)$.

Output Parameters

<i>a</i>	On exit, the factors L and U from the factorization $A = P^* L^* U^* Q$; the unit diagonal elements of L are not stored. If $U(k, k)$ appears to be less than $smin$, $U(k, k)$ is given the value of $smin$, that is giving a nonsingular perturbed system.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The pivot indices: for $1 \leq i \leq n$, row i of the matrix has been interchanged with row $ipiv(i)$.
<i>jpivot</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. The pivot indices: for $1 \leq j \leq n$, column j of the matrix has been interchanged with column $jpivot(j)$.

info

INTEGER.

If *info* = 0, the execution is successful.If *info* = $k > 0$, $U(k, k)$ is likely to produce overflow if we try to solve for x in $A^*x = b$. So U is perturbed to avoid the overflow.

?getf2

Computes the LU factorization of a general m -by- n matrix using partial pivoting with row interchanges (unblocked algorithm).

Syntax

```
call sgetf2( m, n, a, lda, ipiv, info )
call dgetf2( m, n, a, lda, ipiv, info )
call cgetf2( m, n, a, lda, ipiv, info )
call zgetf2( m, n, a, lda, ipiv, info )
```

C:

```
lapack_int LAPACKE_<?>getf2 (int matrix_layout, lapack_int m, lapack_int n, <datatype>
* a, lapack_int lda, lapack_int * ipiv);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the LU factorization of a general m -by- n matrix A using partial pivoting with row interchanges. The factorization has the form

$$A = P^* L^* U$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).

Input Parameters

The data types are given for the Fortran interface.

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a REAL for sgetf2

DOUBLE PRECISION for dgetf2

COMPLEX for cgetf2

DOUBLE COMPLEX for zgetf2.

Array, DIMENSION (*lda,**). Contains the matrix A to be factored. The second dimension of *a* must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

Output Parameters

a Overwritten by L and U . The unit diagonal elements of L are not stored.

ipiv INTEGER.

Array, DIMENSION at least $\max(1, \min(m, n))$.

The pivot indices: for $1 \leq i \leq n$, row i was interchanged with row $ipiv(i)$.

info INTEGER. If $info=0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i > 0$, u_{ii} is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

?gtts2

Solves a system of linear equations with a tridiagonal matrix using the LU factorization computed by ?gttrf.

Syntax

```
call sgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call dgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call cgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
call zgtts2( itrans, n, nrhs, dl, d, du, du2, ipiv, b, ldb )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine solves for X one of the following systems of linear equations with multiple right hand sides:

$A^*X = B$, $A^T*X = B$, or $A^H*X = B$ (for complex matrices only), with a tridiagonal matrix A using the LU factorization computed by ?gttrf.

Input Parameters

ittrans INTEGER. Must be 0, 1, or 2.

Indicates the form of the equations to be solved:

If $ittrans = 0$, then $A^*X = B$ (no transpose).

If $ittrans = 1$, then $A^T*X = B$ (transpose).

If $ittrans = 2$, then $A^H*X = B$ (conjugate transpose).

n INTEGER. The order of the matrix A ($n \geq 0$).

<i>nrhs</i>	INTEGER. The number of right-hand sides, i.e., the number of columns in <i>B</i> ($nrhs \geq 0$).
<i>d1,d,du,du2,b</i>	<p>REAL for sgtts2 DOUBLE PRECISION for dgts2 COMPLEX for cgts2 DOUBLE COMPLEX for zgts2.</p> <p>Arrays: <i>d1</i>(<i>n</i> - 1), <i>d</i>(<i>n</i>), <i>du</i>(<i>n</i> - 1), <i>du2</i>(<i>n</i> - 2), <i>b</i>(<i>ldb</i>, <i>nrhs</i>).</p> <p>The array <i>d1</i> contains the (<i>n</i> - 1) multipliers that define the matrix <i>L</i> from the <i>LU</i> factorization of <i>A</i>.</p> <p>The array <i>d</i> contains the <i>n</i> diagonal elements of the upper triangular matrix <i>U</i> from the <i>LU</i> factorization of <i>A</i>.</p> <p>The array <i>du</i> contains the (<i>n</i> - 1) elements of the first super-diagonal of <i>U</i>.</p> <p>The array <i>du2</i> contains the (<i>n</i> - 2) elements of the second super-diagonal of <i>U</i>.</p> <p>The array <i>b</i> contains the matrix <i>B</i> whose columns are the right-hand sides for the systems of equations.</p>
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> ; must be $ldb \geq \max(1, n)$.
<i>ipiv</i>	<p>INTEGER.</p> <p>Array, DIMENSION (<i>n</i>). The pivot indices array, as returned by ?gttrf.</p>

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
----------	---

?isnan

Tests input for NaN.

Syntax

```
val = sisnan( sin )
val = disnan( din )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This logical routine returns .TRUE. if its argument is NaN, and .FALSE. otherwise.

Input Parameters

<i>sin</i>	REAL for sisnan
------------	------------------------

Input to test for NaN.

din DOUBLE PRECISION for disnan
Input to test for NaN.

Output Parameters

val Logical. Result of the test.

?laisnan

Tests input for NaN.

Syntax

```
val = slaisnan( sin1, sin2 )
val = dlaisnan( din1, din2 )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This logical routine checks for NaNs (NaN stands for 'Not A Number') by comparing its two arguments for inequality. NaN is the only floating-point value where $\text{NaN} \neq \text{NaN}$ returns .TRUE.. To check for NaNs, pass the same variable as both arguments.

This routine is not for general use. It exists solely to avoid over-optimization in ?isnan.

Input Parameters

<i>sin1, sin2</i>	REAL for sisnan
	Two numbers to compare for inequality.
<i>din2, din2</i>	DOUBLE PRECISION for disnan
	Two numbers to compare for inequality.

Output Parameters

val Logical. Result of the comparison.

?labrd

Reduces the first *nb* rows and columns of a general matrix to a bidiagonal form.

Syntax

```
call slabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call dlabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
call clabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
```

```
call zlabrd( m, n, nb, a, lda, d, e, tauq, taup, x, ldx, y, ldy )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine reduces the first nb rows and columns of a general m -by- n matrix A to upper or lower bidiagonal form by an orthogonal/unitary transformation Q^*A^*P , and returns the matrices X and Y which are needed to apply the transformation to the unreduced part of A .

if $m \geq n$, A is reduced to upper bidiagonal form; if $m < n$, to lower bidiagonal form.

The matrices Q and P are represented as products of elementary reflectors: $Q = H(1) * (2) * \dots * H(nb)$, and $P = G(1) * G(2) * \dots * G(nb)$

Each $H(i)$ and $G(i)$ has the form

$H(i) = I - \tau_{aq} v^* v'$ and $G(i) = I - \tau_{ap} u^* u'$

where τ_{aq} and τ_{ap} are scalars, and v and u are vectors.

The elements of the vectors v and u together form the m -by- nb matrix V and the nb -by- n matrix U' which are needed, with X and Y , to apply the transformation to the unreduced part of the matrix, using a block update of the form: $A := A - V^* Y' - X^* U'$.

This is an auxiliary routine called by [?gebrd](#).

Input Parameters

<i>m</i>	INTEGER. The number of rows in the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in A ($n \geq 0$).
<i>nb</i>	INTEGER. The number of leading rows and columns of A to be reduced.
<i>a</i>	REAL for slabrd DOUBLE PRECISION for dlabrd COMPLEX for clabrd DOUBLE COMPLEX for zlabrd. Array $a(lda,*)$ contains the matrix A to be reduced. The second dimension of a must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, m)$.
<i>ldx</i>	INTEGER. The leading dimension of the output array x ; must beat least $\max(1, m)$.
<i>ldy</i>	INTEGER. The leading dimension of the output array y ; must beat least $\max(1, n)$.

Output Parameters

<i>a</i>	On exit, the first nb rows and columns of the matrix are overwritten; the rest of the array is unchanged.
----------	---

if $m \geq n$, elements on and below the diagonal in the first nb columns, with the array τ_{aq} , represent the orthogonal/unitary matrix Q as a product of elementary reflectors; and elements above the diagonal in the first nb rows, with the array τ_{ap} , represent the orthogonal/unitary matrix P as a product of elementary reflectors.

if $m < n$, elements below the diagonal in the first nb columns, with the array τ_{aq} , represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements on and above the diagonal in the first nb rows, with the array τ_{ap} , represent the orthogonal/unitary matrix P as a product of elementary reflectors.

d, e

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors. Arrays, DIMENSION (nb) each. The array d contains the diagonal elements of the first nb rows and columns of the reduced matrix:

$d(i) = a(i,i).$

The array e contains the off-diagonal elements of the first nb rows and columns of the reduced matrix.

τ_{aq}, τ_{ap}

REAL for slabrd

DOUBLE PRECISION for dlabrd

COMPLEX for clabrd

DOUBLE COMPLEX for zlabrd.

Arrays, DIMENSION (nb) each. Contain scalar factors of the elementary reflectors which represent the orthogonal/unitary matrices Q and P , respectively.

x, y

REAL for slabrd

DOUBLE PRECISION for dlabrd

COMPLEX for clabrd

DOUBLE COMPLEX for zlabrd.

Arrays, dimension $x(Idx, nb)$, $y(Idy, nb)$.

The array x contains the m -by- nb matrix X required to update the unreduced part of A .

The array y contains the n -by- nb matrix Y required to update the unreduced part of A .

Application Notes

if $m \geq n$, then for the elementary reflectors $H(i)$ and $G(i)$,

$v(1:i-1) = 0$, $v(i) = 1$, and $v(i:m)$ is stored on exit in $a(i:m, i)$; $u(1:i) = 0$, $u(i+1) = 1$, and $u(i+1:n)$ is stored on exit in $a(i, i+1:n)$;

τ_{aq} is stored in $\tau_{aq}(i)$ and τ_{ap} in $\tau_{ap}(i)$.

if $m < n$,

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+1:m)$ is stored on exit in $a(i+2:m, i)$; $u(1:i-1) = 0$, $u(i) = 1$, and $u(i:n)$ is stored on exit in $a(i, i+1:n)$; τ_{aq} is stored in $\tau_{aq}(i)$ and τ_{ap} in $\tau_{ap}(i)$.

The contents of a on exit are illustrated by the following examples with $nb = 2$:

$m = 6, n = 5 \ (m > n)$

$$\begin{bmatrix} 1 & 1 & u_1 & u_1 & u_1 \\ v_1 & 1 & 1 & u_2 & u_2 \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

$m = 5, n = 6 \ (m < n)$

$$\begin{bmatrix} 1 & u_1 & u_1 & u_1 & u_1 & u_1 \\ 1 & 1 & u_2 & u_2 & u_2 & u_2 \\ v_1 & 1 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \\ v_1 & v_2 & a & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix which is unchanged, v_i denotes an element of the vector defining $H(i)$, and u_i an element of the vector defining $G(i)$.

?lacn2

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

FORTRAN 77:

```
call slacn2( n, v, x, isgn, est, kase, isave )
call dlacn2( n, v, x, isgn, est, kase, isave )
call clacn2( n, v, x, est, kase, isave )
call zlacn2( n, v, x, est, kase, isave )
```

C:

```
lapack_int LAPACKE_slacn2 (lapack_int n, float * v, float * x, lapack_int * isgn, float
* est, lapack_int * kase, lapack_int * isave);
lapack_int LAPACKE_clacn2 (lapack_int n, lapack_complex_float * v, lapack_complex_float
* x, float * est, lapack_int * kase, lapack_int * isave);
lapack_int LAPACKE_dlacn2 (lapack_int n, double * v, double * x, lapack_int * isgn,
double * est, lapack_int * kase, lapack_int * isave);
lapack_int LAPACKE_zlacn2 (lapack_int n, lapack_complex_double * v,
lapack_complex_double * x, double * est, lapack_int * kase, lapack_int * isave);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine estimates the 1-norm of a square, real or complex matrix A . Reverse communication is used for evaluating matrix-vector products.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A ($n \geq 1$).
<i>v, x</i>	<p>REAL for <code>slacn2</code></p> <p>DOUBLE PRECISION for <code>dlacn2</code></p> <p>COMPLEX for <code>clacn2</code></p> <p>DOUBLE COMPLEX for <code>zlacn2</code>.</p> <p>Arrays, DIMENSION (n) each.</p> <p><i>v</i> is a workspace array.</p> <p><i>x</i> is used as input after an intermediate return.</p>
<i>isgn</i>	INTEGER.
	Workspace array, DIMENSION (n), used with real flavors only.
<i>est</i>	<p>REAL for <code>slacn2/clacn2</code></p> <p>DOUBLE PRECISION for <code>dlacn2/zlacn2</code></p> <p>On entry with <i>kase</i> set to 1 or 2, and <i>isave</i>(1) = 1, <i>est</i> must be unchanged from the previous call to the routine.</p>
<i>kase</i>	INTEGER.
	On the initial call to the routine, <i>kase</i> must be set to 0.
<i>isave</i>	<p>INTEGER. Array, DIMENSION (3).</p> <p>Contains variables from the previous call to the routine.</p>

Output Parameters

<i>est</i>	An estimate (a lower bound) for $\text{norm}(A)$.
<i>kase</i>	On an intermediate return, <i>kase</i> is set to 1 or 2, indicating whether <i>x</i> is overwritten by A^*x or $A^{T*}x$ for real flavors and A^*x or $A^{H*}x$ for complex flavors.
	On the final return, <i>kase</i> is set to 0.
<i>v</i>	On the final return, <i>v</i> = A^*w , where <i>est</i> = $\text{norm}(v)/\text{norm}(w)$ (<i>w</i> is not returned).
<i>x</i>	<p>On an intermediate return, <i>x</i> is overwritten by</p> <p>A^*x, if <i>kase</i> = 1,</p> <p>$A^{T*}x$, if <i>kase</i> = 2 (for real flavors),</p> <p>$A^{H*}x$, if <i>kase</i> = 2 (for complex flavors),</p> <p>and the routine must be re-called with all the other parameters unchanged.</p>
<i>isave</i>	This parameter is used to save variables between calls to the routine.

?lacon

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

```
call slacon( n, v, x, isgn, est, kase )
call dlacon( n, v, x, isgn, est, kase )
call clacon( n, v, x, est, kase )
call zlacon( n, v, x, est, kase )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine estimates the 1-norm of a square, real/complex matrix A . Reverse communication is used for evaluating matrix-vector products.

WARNING

The ?lacon routine is not thread-safe. It is deprecated and retained for the backward compatibility only. Use the thread-safe ?lacn2 routine instead.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A ($n \geq 1$).
<i>v</i> , <i>x</i>	REAL for slacon DOUBLE PRECISION for dlacon COMPLEX for clacon DOUBLE COMPLEX for zlacon. Arrays, DIMENSION (n) each. <i>v</i> is a workspace array. <i>x</i> is used as input after an intermediate return.
<i>isgn</i>	INTEGER. Workspace array, DIMENSION (n), used with real flavors only.
<i>est</i>	REAL for slacon/clacon DOUBLE PRECISION for dlacon/zlacon An estimate that with $kase=1$ or 2 should be unchanged from the previous call to ?lacon.
<i>kase</i>	INTEGER.

On the initial call to ?lacon, *kase* should be 0.

Output Parameters

<i>est</i>	REAL for slacon/clacon DOUBLE PRECISION for dlacon/zlacon An estimate (a lower bound) for $\text{norm}(A)$.
<i>kase</i>	On an intermediate return, <i>kase</i> will be 1 or 2, indicating whether <i>x</i> should be overwritten by A^*x or $A^{T*}x$ for real flavors and A^*x or $A^{H*}x$ for complex flavors. On the final return from ?lacon, <i>kase</i> will again be 0.
<i>v</i>	On the final return, $v = A^*w$, where <i>est</i> = $\text{norm}(v) / \text{norm}(w)$ (<i>w</i> is not returned).
<i>x</i>	On an intermediate return, <i>x</i> should be overwritten by A^*x , if <i>kase</i> = 1, $A^{T*}x$, if <i>kase</i> = 2 (for real flavors), $A^{H*}x$, if <i>kase</i> = 2 (for complex flavors), and ?lacon must be re-called with all the other parameters unchanged.

?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

```
call slacpy( uplo, m, n, a, lda, b, ldb )
call dlapcy( uplo, m, n, a, lda, b, ldb )
call clacpy( uplo, m, n, a, lda, b, ldb )
call zlacpy( uplo, m, n, a, lda, b, ldb )
```

C:

```
lapack_int LAPACKE_<?>lacpy (int matrix_layout, char uplo, lapack_int m, lapack_int n,
const <datatype> * a, lapack_int lda, <datatype> * b, lapack_int ldb);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine copies all or part of a two-dimensional matrix *A* to another matrix *B*.

Input Parameters

The data types are given for the Fortran interface.

<i>uplo</i>	CHARACTER*1.
-------------	--------------

Specifies the part of the matrix A to be copied to B .

If $uplo = 'U'$, the upper triangular part of A ;

if $uplo = 'L'$, the lower triangular part of A .

Otherwise, all of the matrix A is copied.

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a REAL for slacpy

DOUBLE PRECISION for dlacpy

COMPLEX for clacpy

DOUBLE COMPLEX for zlacpy.

Array $a(1:da, :)$, contains the m -by- n matrix A .

The second dimension of a must be at least $\max(1, n)$.

If $uplo = 'U'$, only the upper triangle or trapezoid is accessed; if $uplo = 'L'$, only the lower triangle or trapezoid is accessed.

lda INTEGER. The leading dimension of a ; $lda \geq \max(1, m)$.

ldb INTEGER. The leading dimension of the output array b ; $ldb \geq \max(1, m)$.

Output Parameters

b REAL for slacpy

DOUBLE PRECISION for dlacpy

COMPLEX for clacpy

DOUBLE COMPLEX for zlacpy.

Array $b(1:db, :)$, contains the m -by- n matrix B .

The second dimension of b must be at least $\max(1, n)$.

On exit, $B = A$ in the locations specified by $uplo$.

?ladiv

Performs complex division in real arithmetic, avoiding unnecessary overflow.

Syntax

```
call sladiv( a, b, c, d, p, q )
call dladiv( a, b, c, d, p, q )
res = cladiv( x, y )
res = zladiv( x, y )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routines `sladiv/dladiv` perform complex division in real arithmetic as

$$p + iq = \frac{a + ib}{c + id}$$

Complex functions `cladiv/zladiv` compute the result as

$$res = x/y,$$

where x and y are complex. The computation of x / y will not overflow on an intermediary step unless the results overflows.

The algorithm used is due to [Baudin12].

Input Parameters

a, b, c, d	REAL for <code>sladiv</code>
	DOUBLE PRECISION for <code>dladiv</code>
	The scalars a, b, c , and d in the above expression (for real flavors only).
x, y	COMPLEX for <code>cladiv</code>
	DOUBLE COMPLEX for <code>zladiv</code>
	The complex scalars x and y (for complex flavors only).

Output Parameters

p, q	REAL for <code>sladiv</code>
	DOUBLE PRECISION for <code>dladiv</code>
	The scalars p and q in the above expression (for real flavors only).
res	COMPLEX for <code>cladiv</code>
	DOUBLE COMPLEX for <code>zladiv</code>
	Contains the result of division x / y .

?lae2

Computes the eigenvalues of a 2-by-2 symmetric matrix.

Syntax

```
call slae2( a, b, c, rt1, rt2 )
call dlae2( a, b, c, rt1, rt2 )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routines `sla2/dlae2` compute the eigenvalues of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

On return, $rt1$ is the eigenvalue of larger absolute value, and $rt2$ is the eigenvalue of smaller absolute value.

Input Parameters

a, b, c

REAL for `sla2`
DOUBLE PRECISION for `dlae2`

The elements a, b , and c of the 2-by-2 matrix above.

Output Parameters

$rt1, rt2$

REAL for `sla2`
DOUBLE PRECISION for `dlae2`

The computed eigenvalues of larger and smaller absolute value, respectively.

Application Notes

$rt1$ is accurate to a few ulps barring over/underflow. $rt2$ may be inaccurate if there is massive cancellation in the determinant $a*c-b*b$; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute $rt2$ accurately in all cases.

Overflow is possible only if $rt1$ is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds

$underflow_threshold / \text{macheps}$.

?laebz

Computes the number of eigenvalues of a real symmetric tridiagonal matrix which are less than or equal to a given value, and performs other tasks required by the routine ?stebz.

Syntax

```
call slaebz( ijob, nitmax, n, mmax, minp, nbmin, abstol, reltol, pivmin, d, e, e2,
nval, ab, c, mout, nab, work, iwork, info )

call dlaebz( ijob, nitmax, n, mmax, minp, nbmin, abstol, reltol, pivmin, d, e, e2,
nval, ab, c, mout, nab, work, iwork, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?laebz contains the iteration loops which compute and use the function $n(w)$, which is the count of eigenvalues of a symmetric tridiagonal matrix T less than or equal to its argument w . It performs a choice of two types of loops:

- i_{job} =1, followed by
- i_{job} =2: It takes as input a list of intervals and returns a list of sufficiently small intervals whose union contains the same eigenvalues as the union of the original intervals. The input intervals are $(ab(j,1), ab(j,2))$, $j=1, \dots, minp$. The output interval $(ab(j,1), ab(j,2))$ will contain eigenvalues $nab(j, 1)+1, \dots, nab(j, 2)$, where $1 \leq j \leq mout$.
- i_{job} =3: It performs a binary search in each input interval $(ab(j,1), ab(j,2))$ for a point $w(j)$ such that $n(w(j))=nval(j)$, and uses $c(j)$ as the starting point of the search. If such a $w(j)$ is found, then on output $ab(j,1)=ab(j,2)=w$. If no such $w(j)$ is found, then on output $(ab(j,1), ab(j,2))$ will be a small interval containing the point where $n(w)$ jumps through $nval(j)$, unless that point lies outside the initial interval.

Note that the intervals are in all cases half-open intervals, that is, of the form $(a, b]$, which includes b but not a .

To avoid underflow, the matrix should be scaled so that its largest element is no greater than $overflow^{1/2} * overflow^{1/4}$ in absolute value. To assure the most accurate computation of small eigenvalues, the matrix should be scaled to be not much smaller than that, either.

NOTE

In general, the arguments are not checked for unreasonable values.

Input Parameters

- i_{job} INTEGER. Specifies what is to be done:
- = 1: Compute nab for the initial intervals.
 - = 2: Perform bisection iteration to find eigenvalues of T .
 - = 3: Perform bisection iteration to invert $n(w)$, i.e., to find a point which has a specified number of eigenvalues of T to its left. Other values will cause ?laebz to return with $info=-1$.
- $nitmax$ INTEGER. The maximum number of "levels" of bisection to be performed, i.e., an interval of width W will not be made smaller than $2^{-nitmax}W$. If not all intervals have converged after $nitmax$ iterations, then $info$ is set to the number of non-converged intervals.
- n INTEGER. The dimension n of the tridiagonal matrix T . It must be at least 1.

<i>mmax</i>	INTEGER. The maximum number of intervals. If more than <i>mmax</i> intervals are generated, then ?laebz will quit with <i>info</i> = <i>mmax</i> +1.
<i>minp</i>	INTEGER. The initial number of intervals. It may not be greater than <i>mmax</i> .
<i>nbmin</i>	INTEGER. The smallest number of intervals that should be processed using a vector loop. If zero, then only the scalar loop will be used.
<i>abstol</i>	<p>REAL for slaebz</p> <p>DOUBLE PRECISION for dlaebz.</p> <p>The minimum (absolute) width of an interval. When an interval is narrower than <i>abstol</i>, or than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. This must be at least zero.</p>
<i>reltol</i>	<p>REAL for slaebz</p> <p>DOUBLE PRECISION for dlaebz.</p> <p>The minimum relative width of an interval. When an interval is narrower than <i>abstol</i>, or than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. Note: this should always be at least <i>radix*machine epsilon</i>.</p>
<i>pivmin</i>	<p>REAL for slaebz</p> <p>DOUBLE PRECISION for dlaebz.</p> <p>The minimum absolute value of a "pivot" in the Sturm sequence loop. This value must be at least ($\max e(j) ^2 * safe_min$) and at least <i>safe_min</i>, where <i>safe_min</i> is at least the smallest number that can divide one without overflow.</p>
<i>d, e, e2</i>	<p>REAL for slaebz</p> <p>DOUBLE PRECISION for dlaebz.</p> <p>Arrays, dimension (<i>n</i>) each. The array <i>d</i> contains the diagonal elements of the tridiagonal matrix <i>T</i>.</p> <p>The array <i>e</i> contains the off-diagonal elements of the tridiagonal matrix <i>T</i> in positions 1 through <i>n</i>-1. <i>e(n)</i> is arbitrary.</p> <p>The array <i>e2</i> contains the squares of the off-diagonal elements of the tridiagonal matrix <i>T</i>. <i>e2(n)</i> is ignored.</p>
<i>nval</i>	<p>INTEGER.</p> <p>Array, dimension (<i>minp</i>).</p> <p>If <i>ijob</i>=1 or 2, not referenced.</p> <p>If <i>ijob</i>=3, the desired values of <i>n(w)</i>.</p>
<i>ab</i>	<p>REAL for slaebz</p> <p>DOUBLE PRECISION for dlaebz.</p> <p>Array, dimension (<i>mmax</i>,2) The endpoints of the intervals. <i>ab(j,1)</i> is <i>a(j)</i>, the left endpoint of the <i>j</i>-th interval, and <i>ab(j,2)</i> is <i>b(j)</i>, the right endpoint of the <i>j</i>-th interval.</p>

c REAL for `slaebz`
 DOUBLE PRECISION for `dlaebz`.
 Array, dimension (*mmax*)
 If *ijob*=1, ignored.
 If *ijob*=2, workspace.
 If *ijob*=3, then on input *c(j)* should be initialized to the first search point in the binary search.

nab INTEGER.
 Array, dimension (*mmax*,2)
 If *ijob*=2, then on input, *nab(i,j)* should be set. It must satisfy the condition:

$$n(ab(i,1)) \leq nab(i,1) \leq nab(i,2) \leq n(ab(i,2)),$$
 which means that in interval *i* only eigenvalues $nab(i,1)+1, \dots, nab(i,2)$ are considered.
 Usually, $nab(i,j)=n(ab(i,j))$, from a previous call to `?laebz` with *ijob*=1.
 If *ijob*=3, normally, *nab* should be set to some distinctive value(s) before `?laebz` is called.

work REAL for `slaebz`
 DOUBLE PRECISION for `dlaebz`.
 Workspace array, dimension (*mmax*).

iwork INTEGER.
 Workspace array, dimension (*mmax*).

Output Parameters

nval The elements of *nval* will be reordered to correspond with the intervals in *ab*. Thus, *nval(j)* on output will not, in general be the same as *nval(j)* on input, but it will correspond with the interval $(ab(j,1), ab(j,2)]$ on output.

ab The input intervals will, in general, be modified, split, and reordered by the calculation.

mout INTEGER.
 If *ijob*=1, the number of eigenvalues in the intervals.
 If *ijob*=2 or 3, the number of intervals output.
 If *ijob*=3, *mout* will equal *minp*.

nab If *ijob*=1, then on output *nab(i,j)* will be set to $N(ab(i,j))$.
 If *ijob*=2, then on output, *nab(i,j)* will contain $\max(na(k, \min(nb(k), N(ab(i,j))))$, where *k* is the index of the input interval that the output interval $(ab(j,1), ab(j,2)]$ came from, and *na(k)* and *nb(k)* are the input values of *nab(k,1)* and *nab(k,2)*.

If $ijob=3$, then on output, $nab(i,j)$ contains $N(ab(i,j))$, unless $N(w) > nval(i)$ for all search points w , in which case $nab(i,1)$ will not be modified, i.e., the output value will be the same as the input value (modulo reorderings, see $nval$ and ab), or unless $N(w) < nval(i)$ for all search points w , in which case $nab(i,2)$ will not be modified.

info

INTEGER.

If $info = 0$ - all intervals convergedIf $info = 1--mmax$ - the last $info$ interval did not converge.If $info = mmax+1$ - more than $mmax$ intervals were generated

Application Notes

This routine is intended to be called only by other LAPACK routines, thus the interface is less user-friendly. It is intended for two purposes:

(a) finding eigenvalues. In this case, $?laebz$ should have one or more initial intervals set up in ab , and $?laebz$ should be called with $ijob=1$. This sets up nab , and also counts the eigenvalues. Intervals with no eigenvalues would usually be thrown out at this point. Also, if not all the eigenvalues in an interval i are desired, $nab(i,1)$ can be increased or $nab(i,2)$ decreased. For example, set $nab(i,1)=nab(i,2)-1$ to get the largest eigenvalue. $?laebz$ is then called with $ijob=2$ and $mmax$ no smaller than the value of $mout$ returned by the call with $ijob=1$. After this ($ijob=2$) call, eigenvalues $nab(i,1)+1$ through $nab(i,2)$ are approximately $ab(i,1)$ (or $ab(i,2)$) to the tolerance specified by $abstol$ and $reftol$.

(b) finding an interval $(a',b']$ containing eigenvalues $w(f), \dots, w(l)$. In this case, start with a Gershgorin interval (a,b) . Set up ab to contain 2 search intervals, both initially (a,b) . One $nval$ element should contain $f-1$ and the other should contain l , while c should contain a and b , respectively. $nab(i,1)$ should be -1 and $nab(i,2)$ should be $n+1$, to flag an error if the desired interval does not lie in (a,b) . $?laebz$ is then called with $ijob=3$. On exit, if $w(f-1) < w(f)$, then one of the intervals $-- j --$ will have $ab(j,1)=ab(j,2)$ and $nab(j,1)=nab(j,2)=f-1$, while if, to the specified tolerance, $w(f-k)=\dots=w(f+r)$, $k > 0$ and $r \geq 0$, then the interval will have $n(ab(j,1))=nab(j,1)=f-k$ and $n(ab(j,2))=nab(j,2)=f+r$. The cases $w(l) < w(l+1)$ and $w(l-r)=\dots=w(l+k)$ are handled similarly.

?laed0

Used by ?stedc. Computes all eigenvalues and corresponding eigenvectors of an unreduced symmetric tridiagonal matrix using the divide and conquer method.

Syntax

```
call slaed0( icompq, qsiz, n, d, e, q, ldq, qstore, ldqs, work, iwork, info )
call dlaed0( icompq, qsiz, n, d, e, q, ldq, qstore, ldqs, work, iwork, info )
call claed0( qsiz, n, d, e, q, ldq, qstore, ldqs, rwork, iwork, info )
call zlaed0( qsiz, n, d, e, q, ldq, qstore, ldqs, rwork, iwork, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

Real flavors of this routine compute all eigenvalues and (optionally) corresponding eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.

Complex flavors `claed0/zlaed0` compute all eigenvalues of a symmetric tridiagonal matrix which is one diagonal block of those from reducing a dense or band Hermitian matrix and corresponding eigenvectors of the dense or band matrix.

Input Parameters

<code>icompq</code>	INTEGER. Used with real flavors only. If <code>icompq</code> = 0, compute eigenvalues only. If <code>icompq</code> = 1, compute eigenvectors of original dense symmetric matrix also. On entry, the array q must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form. If <code>icompq</code> = 2, compute eigenvalues and eigenvectors of the tridiagonal matrix.
<code>qsiz</code>	INTEGER. The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if <code>icompq</code> = 1).
<code>n</code>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<code>d, e, rwork</code>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: $d(*)$ contains the main diagonal of the tridiagonal matrix. The dimension of d must be at least $\max(1, n)$. $e(*)$ contains the off-diagonal elements of the tridiagonal matrix. The dimension of e must be at least $\max(1, n-1)$. $rwork(*)$ is a workspace array used in complex flavors only. The dimension of $rwork$ must be at least $(1 + 3n + 2n\log_2(n) + 3n^2)$, where $\log_2(n)$ = smallest integer k such that $2^k \geq n$.
<code>q, qstore</code>	REAL for <code>slaed0</code> DOUBLE PRECISION for <code>dlaed0</code> COMPLEX for <code>claed0</code> DOUBLE COMPLEX for <code>zlaed0</code> . Arrays: $q(ldq, *)$, $qstore(ldq, *)$. The second dimension of these arrays must be at least $\max(1, n)$. <i>For real flavors:</i> If <code>icompq</code> = 0, array q is not referenced. If <code>icompq</code> = 1, on entry, q is a subset of the columns of the orthogonal matrix used to reduce the full matrix to tridiagonal form corresponding to the subset of the full matrix which is being decomposed at this time.

If $i\text{compq} = 2$, on entry, q will be the identity matrix. The array $qstore$ is a workspace array referenced only when $i\text{compq} = 1$. Used to store parts of the eigenvector matrix when the updating matrix multiplies take place.

For complex flavors:

On entry, q must contain an $qsiz$ -by- n matrix whose columns are unitarily orthonormal. It is a part of the unitary matrix that reduces the full dense Hermitian matrix to a (reducible) symmetric tridiagonal matrix. The array $qstore$ is a workspace array used to store parts of the eigenvector matrix when the updating matrix multiplies take place.

ldq

INTEGER. The leading dimension of the array q ; $ldq \geq \max(1, n)$.

$ldqs$

INTEGER. The leading dimension of the array $qstore$; $ldqs \geq \max(1, n)$.

$work$

REAL for `slaed0`

DOUBLE PRECISION for `dlaed0`.

Workspace array, used in real flavors only.

If $i\text{compq} = 0$ or 1 , the dimension of $work$ must be at least $(1 + 3n + 2n\log_2(n) + 3n^2)$, where $\log_2(n) =$ smallest integer k such that $2^k \geq n$.

If $i\text{compq} = 2$, the dimension of $work$ must be at least $(4n + n^2)$.

$iwork$

INTEGER.

Workspace array.

For real flavors, if $i\text{compq} = 0$ or 1 , and for complex flavors, the dimension of $iwork$ must be at least $(6 + 6n + 5n\log_2(n))$.

For real flavors, if $i\text{compq} = 2$, the dimension of $iwork$ must be at least $(3 + 5n)$.

Output Parameters

d

On exit, contains eigenvalues in ascending order.

e

On exit, the array is destroyed.

q

If $i\text{compq} = 2$, on exit, q contains the eigenvectors of the tridiagonal matrix.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

If $info = i > 0$, the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $i/(n+1)$ through $\text{mod}(i, n+1)$.

?laed1

Used by sstedc/dstedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is tridiagonal.

Syntax

```
call slaed1( n, d, q, ldq, indxq, rho, cutpnt, work, iwork, info )
call dlaed1( n, d, q, ldq, indxq, rho, cutpnt, work, iwork, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?laed1 computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and eigenvectors of a tridiagonal matrix. ?laed7 handles the case in which eigenvalues only or eigenvalues and eigenvectors of a full symmetric matrix (which was reduced to tridiagonal form) are desired.

$$T = Q(\text{in}) * (D(\text{in}) + \rho Z * Z^T) * Q^T(\text{in}) = Q(\text{out}) * D(\text{out}) * Q^T(\text{out})$$

where $Z = Q^T u$, u is a vector of length n with ones in the cutpnt and $(\text{cutpnt}+1)$ -th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in Q , and the eigenvalues are in D . The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine ?laed2.

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine ?laed4 (as called by ?laed3). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

n INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).

d, q, work REAL for slaed1

DOUBLE PRECISION for dlaed1.

Arrays:

d()* contains the eigenvalues of the rank-1-perturbed matrix. The dimension of *d* must be at least $\max(1, n)$.

*q(ldq, *)* contains the eigenvectors of the rank-1-perturbed matrix. The second dimension of *q* must be at least $\max(1, n)$.

work()* is a workspace array, dimension at least $(4n+n^2)$.

ldq INTEGER. The leading dimension of the array *q*; $ldq \geq \max(1, n)$.

<i>indxq</i>	INTEGER. Array, dimension (<i>n</i>). On entry, the permutation which separately sorts the two subproblems in <i>d</i> into ascending order.
<i>rho</i>	REAL for <code>slaed1</code> DOUBLE PRECISION for <code>dlaed1</code> . The subdiagonal entry used to create the rank-1 modification. This parameter can be modified by <code>?laed2</code> , where it is input/output.
<i>cutpnt</i>	INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1, n) \leq \text{cutpnt} \leq n/2$.
<i>iwork</i>	INTEGER. Workspace array, dimension (4 <i>n</i>).

Output Parameters

<i>d</i>	On exit, contains the eigenvalues of the repaired matrix.
<i>q</i>	On exit, <i>q</i> contains the eigenvectors of the repaired tridiagonal matrix.
<i>indxq</i>	On exit, contains the permutation which will reintegrate the subproblems back into sorted order, that is, <i>d</i> (<i>indxq</i> (<i>i</i> = 1, <i>n</i>)) will be in ascending order.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, an eigenvalue did not converge.

?laed2

Used by `sstedc/dstedc`. Merges eigenvalues and deflates secular equation. Used when the original matrix is tridiagonal.

Syntax

```
call slaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda, w, q2, indx, indxc, indxp, coltyp, info )
call dlaed2( k, n, n1, d, q, ldq, indxq, rho, z, dlamda, w, q2, indx, indxc, indxp, coltyp, info )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine ?laed2 merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny entry in the z vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

k	INTEGER. The number of non-deflated eigenvalues, and the order of the related secular equation ($0 \leq k \leq n$).
n	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
$n1$	INTEGER. The location of the last eigenvalue in the leading sub-matrix; $\min(1, n) \leq n1 \leq n/2$.
d, q, z	REAL for slaed2 DOUBLE PRECISION for dlaed2. Arrays: $d(*)$ contains the eigenvalues of the two submatrices to be combined. The dimension of d must be at least $\max(1, n)$. $q(ldq, *)$ contains the eigenvectors of the two submatrices in the two square blocks with corners at $(1,1), (n1,n1)$ and $(n1+1,n1+1), (n,n)$. The second dimension of q must be at least $\max(1, n)$. $z(*)$ contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix). ldq INTEGER. The leading dimension of the array q ; $ldq \geq \max(1, n)$. $indxq$ INTEGER. Array, dimension (n). On entry, the permutation which separately sorts the two subproblems in d into ascending order. Note that elements in the second half of this permutation must first have $n1$ added to their values. rho REAL for slaed2 DOUBLE PRECISION for dlaed2. On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined. $indx, indxp$ INTEGER. Workspace arrays, dimension (n) each. Array $indx$ contains the permutation used to sort the contents of d into ascending order. Array $indxp$ contains the permutation used to place deflated values of d at the end of the array. $indx(1:k)$ points to the nondeflated d -values and $indx(k+1:n)$ points to the deflated eigenvalues. $coltyp$ INTEGER. Workspace array, dimension (n). During execution, a label which will indicate which of the following types a column in the $q2$ matrix is:

- 1 : non-zero in the upper half only;
- 2 : dense;
- 3 : non-zero in the lower half only;
- 4 : deflated.

Output Parameters

<i>d</i>	On exit, <i>d</i> contains the trailing ($n-k$) updated eigenvalues (those which were deflated) sorted into increasing order.
<i>q</i>	On exit, <i>q</i> contains the trailing ($n-k$) updated eigenvectors (those which were deflated) in its last $n-k$ columns.
<i>z</i>	On exit, <i>z</i> content is destroyed by the updating process.
<i>indxq</i>	Destroyed on exit.
<i>rho</i>	On exit, <i>rho</i> has been modified to the value required by ?laed3.
<i>dlamda, w, q2</i>	<p>REAL for dlaed2</p> <p>DOUBLE PRECISION for dlaed2.</p> <p>Arrays: <i>dlamda</i>(n), <i>w</i>(n), <i>q2</i>($n1^2 + (n-n1)^2$).</p> <p>The array <i>dlamda</i> contains a copy of the first k eigenvalues which is used by ?laed3 to form the secular equation.</p> <p>The array <i>w</i> contains the first k values of the final deflation-altered <i>z</i>-vector which is passed to ?laed3.</p> <p>The array <i>q2</i> contains a copy of the first k eigenvectors which is used by ?laed3 in a matrix multiply (sgemm/dgemm) to solve for the new eigenvectors.</p>
<i>indxk</i>	INTEGER. Array, dimension (n).
	The permutation used to arrange the columns of the deflated <i>q</i> matrix into three groups: the first group contains non-zero elements only at and above $n1$, the second contains non-zero elements only below $n1$, and the third is dense.
<i>coltyp</i>	On exit, <i>coltyp</i> (<i>i</i>) is the number of columns of type <i>i</i> , for <i>i</i> =1 to 4 only (see the definition of types in the description of <i>coltyp</i> in <i>Input Parameters</i>).
<i>info</i>	<p>INTEGER.</p> <p>If <i>info</i> = 0, the execution is successful.</p> <p>If <i>info</i> = -<i>i</i>, the <i>i</i>-th parameter had an illegal value.</p>

?laed3

Used by sstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is tridiagonal.

Syntax

```
call slaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx, ctot, w, s, info )
call dlaed3( k, n, n1, d, q, ldq, rho, dlamda, q2, indx, ctot, w, s, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?laed3 finds the roots of the secular equation, as defined by the values in d , w , and ρ , between 1 and k .

It makes the appropriate calls to ?laed4 and then updates the eigenvectors by multiplying the matrix of eigenvectors of the pair of eigensystems being combined by the matrix of eigenvectors of the k -by- k system which is solved here.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray X-MP, Cray Y-MP, Cray C-90, or Cray-2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but none are known.

Input Parameters

k	INTEGER. The number of terms in the rational function to be solved by ?laed4 ($k \geq 0$).
n	INTEGER. The number of rows and columns in the q matrix. $n \geq k$ (deflation may result in $n > k$).
$n1$	INTEGER. The location of the last eigenvalue in the leading sub-matrix; $\min(1, n) \leq n1 \leq n/2$.
q	REAL for slaed3 DOUBLE PRECISION for dlaed3. Array $q(ldq, *)$. The second dimension of q must be at least $\max(1, n)$. Initially, the first k columns of this array are used as workspace.
ldq	INTEGER. The leading dimension of the array q ; $ldq \geq \max(1, n)$.
ρ	REAL for slaed3 DOUBLE PRECISION for dlaed3. The value of the parameter in the rank one update equation. $\rho \geq 0$ required.
$dlamda, q2, w$	REAL for slaed3 DOUBLE PRECISION for dlaed3. Arrays: $dlamda(k)$, $q2(1:ldq2, *)$, $w(k)$. The first k elements of the array $dlamda$ contain the old roots of the deflated updating problem. These are the poles of the secular equation.

The first k columns of the array $q2$ contain the non-deflated eigenvectors for the split problem. The second dimension of $q2$ must be at least $\max(1, n)$.

The first k elements of the array w contain the components of the deflation-adjusted updating vector.

indx

INTEGER. Array, dimension (n).

The permutation used to arrange the columns of the deflated q matrix into three groups (see ?laed2).

The rows of the eigenvectors found by ?laed4 must be likewise permuted before the matrix multiply can take place.

ctot

INTEGER. Array, dimension (4).

A count of the total number of the various types of columns in q , as described in *indx*. The fourth column type is any column which has been deflated.

s

REAL for slaed3

DOUBLE PRECISION for dlaed3.

Workspace array, dimension $(n1+1)*k$.

Will contain the eigenvectors of the repaired matrix which will be multiplied by the previously accumulated eigenvectors to update the system.

Output Parameters

d

REAL for slaed3

DOUBLE PRECISION for dlaed3.

Array, dimension at least $\max(1, n)$.

d(i) contains the updated eigenvalues for $1 \leq i \leq k$.

q

On exit, the columns 1 to k of q contain the updated eigenvectors.

dlambda

May be changed on output by having lowest order bit set to zero on Cray X-MP, Cray Y-MP, Cray-2, or Cray C-90, as described above.

w

Destroyed on exit.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the i -th parameter had an illegal value.

If *info* = 1, an eigenvalue did not converge.

?laed4

Used by sstedc/dstedc. Finds a single root of the secular equation.

Syntax

```
call slaed4( n, i, d, z, delta, rho, dlam, info )
call dlaed4( n, i, d, z, delta, rho, dlam, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This routine computes the i -th updated eigenvalue of a symmetric rank-one modification to a diagonal matrix whose elements are given in the array d , and that

$D(i) < D(j)$ for $i < j$

and that $\rho > 0$. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

$\text{diag}(D) + \rho * Z * \text{transpose}(Z)$.

where we assume the Euclidean norm of Z is 1.

The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

n	INTEGER. The length of all arrays.
i	INTEGER. The index of the eigenvalue to be computed; $1 \leq i \leq n$.
d, z	REAL for slaed4 DOUBLE PRECISION for dlaed4 Arrays, dimension (n) each. The array d contains the original eigenvalues. It is assumed that they are in order, $d(i) < d(j)$ for $i < j$. The array z contains the components of the updating vector Z .
ρ	REAL for slaed4 DOUBLE PRECISION for dlaed4 The scalar in the symmetric updating formula.

Output Parameters

δ	REAL for slaed4 DOUBLE PRECISION for dlaed4 Array, dimension (n). If $n \neq 1$, δ contains $(d(j) - \lambda_i)$ in its j -th component. If $n = 1$, then $\delta(1) = 1$. The vector δ contains the information necessary to construct the eigenvectors.
λ	REAL for slaed4 DOUBLE PRECISION for dlaed4

The computed λ_i , the i -th updated eigenvalue.

info INTEGER.

If $info = 0$, the execution is successful.

If $info = 1$, the updating process failed.

?laed5

Used by `sstedc/dstedc`. Solves the 2-by-2 secular equation.

Syntax

```
call slaed5( i, d, z, delta, rho, dlam )
call dlaed5( i, d, z, delta, rho, dlam )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine computes the i -th eigenvalue of a symmetric rank-one modification of a 2-by-2 diagonal matrix $\text{diag}(D) + \rho Z * \text{transpose}(Z)$.

The diagonal elements in the array D are assumed to satisfy

$D(i) < D(j)$ for $i < j$.

We also assume $\rho > 0$ and that the Euclidean norm of the vector Z is one.

Input Parameters

i INTEGER. The index of the eigenvalue to be computed;
 $1 \leq i \leq 2$.

d, z REAL for `slaed5`
 DOUBLE PRECISION for `dlaed5`
 Arrays, dimension (2) each. The array d contains the original eigenvalues.
 It is assumed that $d(1) < d(2)$.
 The array z contains the components of the updating vector.

rho REAL for `slaed5`
 DOUBLE PRECISION for `dlaed5`
 The scalar in the symmetric updating formula.

Output Parameters

delta REAL for `slaed5`
 DOUBLE PRECISION for `dlaed5`

Array, dimension (2).

The vector *delta* contains the information necessary to construct the eigenvectors.

dlam REAL for slaed5

DOUBLE PRECISION for dlaed5

The computed *lambda_i*, the *i*-th updated eigenvalue.

?laed6

Used by sstedc/dstedc. Computes one Newton step
in solution of the secular equation.

Syntax

```
call slaed6( kniter, orgati, rho, d, z, finit, tau, info )
call dlaed6( kniter, orgati, rho, d, z, finit, tau, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the positive or negative root (closest to the origin) of

$$f(x) = rho + \frac{z(1)}{d(1) - x} + \frac{z(2)}{d(2) - x} + \frac{z(3)}{d(3) - x}$$

It is assumed that if *orgati* = .TRUE. the root is between *d*(2) and *d*(3); otherwise it is between *d*(1) and *d*(2). This routine is called by ?laed4 when necessary. In most cases, the root sought is the smallest in magnitude, though it might not be in some extremely rare situations.

Input Parameters

kniter

INTEGER.

Refer to ?laed4 for its significance.

orgati

LOGICAL.

If *orgati* = .TRUE., the needed root is between *d*(2) and *d*(3); otherwise it is between *d*(1) and *d*(2). See ?laed4 for further details.

rho

REAL for slaed6

DOUBLE PRECISION for dlaed6

Refer to the equation for *f*(*x*) above.

d, z

REAL for slaed6

DOUBLE PRECISION for dlaed6

Arrays, dimension (3) each.

The array d satisfies $d(1) < d(2) < d(3)$.

Each of the elements in the array z must be positive.

finit

REAL for `slaed6`

DOUBLE PRECISION for `dlaed6`

The value of $f(x)$ at 0. It is more accurate than the one evaluated inside this routine (if someone wants to do so).

Output Parameters

tau

REAL for `slaed6`

DOUBLE PRECISION for `dlaed6`

The root of the equation for $f(x)$.

info

INTEGER.

If $info = 0$, the execution is successful.

If $info = 1$, failure to converge.

?laed7

Used by ?stedc. Computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. Used when the original matrix is dense.

Syntax

```
call slaed7( icompq, n, qsiz, tlvls, curlvl, curpbm, d, q, ldq, indxq, rho, cutpnt,  

qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, iwork, info )  

call dlaed7( icompq, n, qsiz, tlvls, curlvl, curpbm, d, q, ldq, indxq, rho, cutpnt,  

qstore, qptr, prmptr, perm, givptr, givcol, givnum, work, iwork, info )  

call claed7( n, cutpnt, qsiz, tlvls, curlvl, curpbm, d, q, ldq, rho, indxq, qstore,  

qptr, prmptr, perm, givptr, givcol, givnum, work, rwork, iwork, info )  

call zlaed7( n, cutpnt, qsiz, tlvls, curlvl, curpbm, d, q, ldq, rho, indxq, qstore,  

qptr, prmptr, perm, givptr, givcol, givnum, work, rwork, iwork, info )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine ?laed7 computes the updated eigensystem of a diagonal matrix after modification by a rank-one symmetric matrix. This routine is used only for the eigenproblem which requires all eigenvalues and optionally eigenvectors of a dense symmetric/Hermitian matrix that has been reduced to tridiagonal form. For real flavors, `slaed1/dlaed1` handles the case in which all eigenvalues and eigenvectors of a symmetric tridiagonal matrix are desired.

$$T = Q(\text{in}) * (D(\text{in}) + \rho Z^T Z) * Q^T(\text{in}) = Q(\text{out}) * D(\text{out}) * Q^T(\text{out}) \text{ for real flavors, or}$$

$T = Q(\text{in}) * (D(\text{in}) + \rho Z^* Z^H) * Q^H(\text{in}) = Q(\text{out}) * D(\text{out}) * Q^H(\text{out})$ for complex flavors

where $Z = Q^T * u$ for real flavors and $Z = Q^H * u$ for complex flavors, u is a vector of length n with ones in the cutpnt and $(\text{cutpnt} + 1)$ -th elements and zeros elsewhere. The eigenvectors of the original matrix are stored in Q , and the eigenvalues are in D . The algorithm consists of three stages:

The first stage consists of deflating the size of the problem when there are multiple eigenvalues or if there is a zero in the z vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `slaed8/dlaed8` (for real flavors) or by the routine `slaed2/dlaed2` (for complex flavors).

The second stage consists of calculating the updated eigenvalues. This is done by finding the roots of the secular equation via the routine `?laed4` (as called by `?laed9` or `?laed3`). This routine also calculates the eigenvectors of the current problem.

The final stage consists of computing the updated eigenvectors directly using the updated eigenvalues. The eigenvectors for the current problem are multiplied with the eigenvectors from the overall problem.

Input Parameters

<i>icompq</i>	INTEGER. Used with real flavors only. If <i>icompq</i> = 0, compute eigenvalues only. If <i>icompq</i> = 1, compute eigenvectors of original dense symmetric matrix also. On entry, the array <i>q</i> must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
<i>n</i>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<i>cutpnt</i>	INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1, n) \leq \text{cutpnt} \leq n$.
<i>qsiz</i>	INTEGER. The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if <i>icompq</i> = 1).
<i>tlvl</i>	INTEGER. The total number of merging levels in the overall divide and conquer tree.
<i>curlvl</i>	INTEGER. The current level in the overall merge routine, $0 \leq \text{curlvl} \leq \text{tlvl}$.
<i>curpbm</i>	INTEGER. The current problem in the current level in the overall merge routine (counting from upper left to lower right).
<i>d</i>	REAL for <code>slaed7/claed7</code> DOUBLE PRECISION for <code>dlaed7/zlaed7</code> . Array, dimension at least $\max(1, n)$. Array <i>d</i> (*) contains the eigenvalues of the rank-1-perturbed matrix.
<i>q, work</i>	REAL for <code>slaed7</code> DOUBLE PRECISION for <code>dlaed7</code> COMPLEX for <code>claed7</code>

DOUBLE COMPLEX **for** zlaed7.

Arrays:

q(*ldq*, *) contains the eigenvectors of the rank-1-perturbed matrix. The second dimension of *q* must be at least $\max(1, n)$.

work(*) is a workspace array, dimension at least $(3n+2*qsiz*n)$ for real flavors and at least $(qsiz*n)$ for complex flavors.

ldq

INTEGER. The leading dimension of the array *q*; $ldq \geq \max(1, n)$.

indxq

INTEGER. Array, dimension (*n*).

Contains the permutation that separately sorts the two sub-problems in *d* into ascending order.

rho

REAL **for** slaed7 /claed7

DOUBLE PRECISION **for** dlaed7/zlaed7.

The subdiagonal element used to create the rank-1 modification.

qstore

REAL **for** slaed7/claed7

DOUBLE PRECISION **for** dlaed7/zlaed7.

Array, dimension (n^2+1). Serves also as output parameter.

Stores eigenvectors of submatrices encountered during divide and conquer, packed together. *qptr* points to beginning of the submatrices.

qptr

INTEGER. Array, dimension (*n*+2). Serves also as output parameter. List of indices pointing to beginning of submatrices stored in *qstore*. The submatrices are numbered starting at the bottom left of the divide and conquer tree, from left to right and bottom to top.

prmptr, perm, givptr

INTEGER. Arrays, dimension ($n \log_2 n$) each.

The array *prmptr*(*) contains a list of pointers which indicate where in *perm* a level's permutation is stored. *prmptr*(*i*+1) - *prmptr*(*i*) indicates the size of the permutation and also the size of the full, non-deflated problem.

The array *perm*(*) contains the permutations (from deflation and sorting) to be applied to each eigenblock. This parameter can be modified by ?laed8, where it is output.

The array *givptr*(*) contains a list of pointers which indicate where in *givcol* a level's Givens rotations are stored. *givptr*(*i*+1) - *givptr*(*i*) indicates the number of Givens rotations.

givcol

INTEGER. Array, dimension (2, $n \log_2 n$).

Each pair of numbers indicates a pair of columns to take place in a Givens rotation.

givnum

REAL **for** slaed7/claed7

DOUBLE PRECISION **for** dlaed7/zlaed7.

Array, dimension (2, $n \log_2 n$).

Each number indicates the S value to be used in the corresponding Givens rotation.

<i>iwork</i>	INTEGER. Workspace array, dimension ($4n$).
<i>rwork</i>	REAL for <code>claed7</code> DOUBLE PRECISION for <code>zlaed7</code> . Workspace array, dimension ($3n+2qsiz*n$). Used in complex flavors only.

Output Parameters

<i>d</i>	On exit, contains the eigenvalues of the repaired matrix.
<i>q</i>	On exit, <i>q</i> contains the eigenvectors of the repaired tridiagonal matrix.
<i>indxq</i>	INTEGER. Array, dimension (n). Contains the permutation that reintegrates the subproblems back into a sorted order, that is, $d(indxq(i = 1, n))$ will be in the ascending order.
<i>rho</i>	This parameter can be modified by <code>?laed8</code> , where it is input/output.
<i>prmptr, perm, givptr</i>	INTEGER. Arrays, dimension ($n \log_2 n$) each. The array <i>prmptr</i> contains an updated list of pointers. The array <i>perm</i> contains an updated permutation. The array <i>givptr</i> contains an updated list of pointers.
<i>givcol</i>	This parameter can be modified by <code>?laed8</code> , where it is output.
<i>givnum</i>	This parameter can be modified by <code>?laed8</code> , where it is output.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value. If <i>info</i> = 1, an eigenvalue did not converge.

?laed8

Used by `?stedc`. Merges eigenvalues and deflates secular equation. Used when the original matrix is dense.

Syntax

```
call slaed8( icompq, k, n, qsiz, d, q, ldq, indxq, rho, cutpnt, z, dlamda, q2, ldq2, w, perm, givptr, givcol, givnum, indxpx, indx, info )  
call dlaed8( icompq, k, n, qsiz, d, q, ldq, indxq, rho, cutpnt, z, dlamda, q2, ldq2, w, perm, givptr, givcol, givnum, indxpx, indx, info )
```

```
call claed8( k, n, qsiz, q, ldq, d, rho, cutpnt, z, dlamda, q2, ldq2, w, indx, indxq, perm, givptr, givcol, givnum, info )
call zlaed8( k, n, qsiz, q, ldq, d, rho, cutpnt, z, dlamda, q2, ldq2, w, indx, indxq, perm, givptr, givcol, givnum, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine merges the two sets of eigenvalues together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more eigenvalues are close together or if there is a tiny element in the z vector. For each such occurrence the order of the related secular equation problem is reduced by one.

Input Parameters

<i>icompq</i>	INTEGER. Used with real flavors only. If <i>icompq</i> = 0, compute eigenvalues only. If <i>icompq</i> = 1, compute eigenvectors of original dense symmetric matrix also. On entry, the array <i>q</i> must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.
<i>n</i>	INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).
<i>cutpnt</i>	INTEGER. The location of the last eigenvalue in the leading sub-matrix. $\min(1, n) \leq \text{cutpnt} \leq n$.
<i>qsiz</i>	INTEGER. The dimension of the orthogonal/unitary matrix used to reduce the full matrix to tridiagonal form; $qsiz \geq n$ (for real flavors, $qsiz \geq n$ if <i>icompq</i> = 1).
<i>d, z</i>	REAL for slaed8/claed8 DOUBLE PRECISION for dlaed8/zlaed8. Arrays, dimension at least $\max(1, n)$ each. The array <i>d</i> (*) contains the eigenvalues of the two submatrices to be combined. On entry, <i>z</i> (*) contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix). The contents of <i>z</i> are destroyed by the updating process.
<i>q</i>	REAL for slaed8 DOUBLE PRECISION for dlaed8 COMPLEX for claed8 DOUBLE COMPLEX for zlaed8. Array

$q(ldq, *)$. The second dimension of q must be at least $\max(1, n)$. On entry, q contains the eigenvectors of the partially solved system which has been previously updated in matrix multiplies with other partially solved eigensystems.

For real flavors, If $i_compq = 0$, q is not referenced.

ldq INTEGER. The leading dimension of the array q ; $ldq \geq \max(1, n)$.

$ldq2$ INTEGER. The leading dimension of the output array $q2$; $ldq2 \geq \max(1, n)$.

$indxq$ INTEGER. Array, dimension (n).

The permutation that separately sorts the two sub-problems in d into ascending order. Note that elements in the second half of this permutation must first have $cutpt$ added to their values in order to be accurate.

rho REAL for `slaed8/claed8`

DOUBLE PRECISION for `dlaed8/zlaed8`.

On entry, the off-diagonal element associated with the rank-1 cut which originally split the two submatrices which are now being recombined.

Output Parameters

k INTEGER. The number of non-deflated eigenvalues, and the order of the related secular equation.

d On exit, contains the trailing $(n-k)$ updated eigenvalues (those which were deflated) sorted into increasing order.

z On exit, the updating process destroys the contents of z .

q On exit, q contains the trailing $(n-k)$ updated eigenvectors (those which were deflated) in its last $(n-k)$ columns.

$indxq$ INTEGER. Array, dimension (n).

The permutation of merged eigenvalues set.

rho On exit, rho has been modified to the value required by `?laed3`.

$dlamda, w$ REAL for `slaed8/claed8`

DOUBLE PRECISION for `dlaed8/zlaed8`.

Arrays, dimension (n) each. The array $dlamda(*)$ contains a copy of the first k eigenvalues which will be used by `?laed3` to form the secular equation.

The array $w(*)$ will hold the first k values of the final deflation-altered z -vector and will be passed to `?laed3`.

$q2$ REAL for `slaed8`

DOUBLE PRECISION for `dlaed8`

COMPLEX for `claed8`

DOUBLE COMPLEX for zlaed8.

Array

$q2(ldq2, *)$. The second dimension of $q2$ must be at least $\max(1, n)$.

Contains a copy of the first k eigenvectors which will be used by slaed7/dlaed7 in a matrix multiply (sgemm/dgemm) to update the new eigenvectors. For real flavors, If $icompq = 0$, $q2$ is not referenced.

$indx_p, indx$

INTEGER. Workspace arrays, dimension (n) each.

The array $indx_p(*)$ will contain the permutation used to place deflated values of d at the end of the array. On output, $indx_p(1:k)$ points to the nondeflated d -values and $indx_p(k+1:n)$ points to the deflated eigenvalues.

The array $indx(*)$ will contain the permutation used to sort the contents of d into ascending order.

$perm$

INTEGER. Array, dimension (n).

Contains the permutations (from deflation and sorting) to be applied to each eigenblock.

$givptr$

INTEGER. Contains the number of Givens rotations which took place in this subproblem.

$givcol$

INTEGER. Array, dimension (2, n).

Each pair of numbers indicates a pair of columns to take place in a Givens rotation.

$givnum$

REAL for slaed8/claed8

DOUBLE PRECISION for dlaed8/zlaed8.

Array, dimension (2, n).

Each number indicates the S value to be used in the corresponding Givens rotation.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -i$, the i -th parameter had an illegal value.

?laed9

Used by sstedc/dstedc. Finds the roots of the secular equation and updates the eigenvectors. Used when the original matrix is dense.

Syntax

```
call slaed9( k, kstart, kstop, n, d, q, ldq, rho, dlamda, w, s, lds, info )
call dlaed9( k, kstart, kstop, n, d, q, ldq, rho, dlamda, w, s, lds, info )
```

Include Files

- Fortran: mkl.fi

- C: mkl.h

Description

The routine finds the roots of the secular equation, as defined by the values in d , z , and ρ , between $kstart$ and $kstop$. It makes the appropriate calls to `slaed4/dlaed4` and then stores the new matrix of eigenvectors for use in calculating the next level of z vectors.

Input Parameters

k	INTEGER. The number of terms in the rational function to be solved by <code>slaed4/dlaed4</code> ($k \geq 0$).
$kstart, kstop$	INTEGER. The updated eigenvalues $\lambda(i)$, $kstart \leq i \leq kstop$ are to be computed. $1 \leq kstart \leq kstop \leq k$.
n	INTEGER. The number of rows and columns in the Q matrix. $n \geq k$ (deflation may result in $n > k$).
q	REAL for <code>slaed9</code> DOUBLE PRECISION for <code>dlaed9</code> . Workspace array, dimension ($ldq, *$). The second dimension of q must be at least $\max(1, n)$.
ldq	INTEGER. The leading dimension of the array q ; $ldq \geq \max(1, n)$.
ρ	REAL for <code>slaed9</code> DOUBLE PRECISION for <code>dlaed9</code> The value of the parameter in the rank one update equation. $\rho \geq 0$ required.
$dlamda, w$	REAL for <code>slaed9</code> DOUBLE PRECISION for <code>dlaed9</code> Arrays, dimension (k) each. The first k elements of the array $dlamda(*)$ contain the old roots of the deflated updating problem. These are the poles of the secular equation. The first k elements of the array $w(*)$ contain the components of the deflation-adjusted updating vector.
lds	INTEGER. The leading dimension of the output array s ; $lds \geq \max(1, k)$.

Output Parameters

d	REAL for <code>slaed9</code> DOUBLE PRECISION for <code>dlaed9</code> Array, dimension (n). Elements in $d(i)$ are not referenced for $1 \leq i < kstart$ or $kstop < i \leq n$.
s	REAL for <code>slaed9</code>

DOUBLE PRECISION for `dlaed9`.

`Array, dimension (lds, *)`.

The second dimension of `s` must be at least `max(1, k)`. Will contain the eigenvectors of the repaired matrix which will be stored for subsequent `z` vector calculation and multiplied by the previously accumulated eigenvectors to update the system.

`dlamda`

On exit, the value is modified to make sure all `dlamda(i) - dlamda(j)` can be computed with high relative accuracy, barring overflow and underflow.

`w`

Destroyed on exit.

`info`

INTEGER.

If `info = 0`, the execution is successful.

If `info = -i`, the i -th parameter had an illegal value. If `info = 1`, the eigenvalue did not converge.

?laeda

Used by ?stedc. Computes the Z vector determining the rank-one modification of the diagonal matrix. Used when the original matrix is dense.

Syntax

```
call slaeda( n, tlvl, curlvl, curpbm, prmprt, perm, givptr, givcol, givnum, q, qptr,
z, ztemp, info )
```

```
call dlaeda( n, tlvl, curlvl, curpbm, prmprt, perm, givptr, givcol, givnum, q, qptr,
z, ztemp, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?laeda computes the Z vector corresponding to the merge step in the `curlvl`-th step of the merge process with `tlvl` steps for the `curpbm`-th problem.

Input Parameters

`n` INTEGER. The dimension of the symmetric tridiagonal matrix ($n \geq 0$).

`tlvl` INTEGER. The total number of merging levels in the overall divide and conquer tree.

`curlvl` INTEGER. The current level in the overall merge routine, $0 \leq curlvl \leq tlvl$.

`curpbm` INTEGER. The current problem in the current level in the overall merge routine (counting from upper left to lower right).

*prmptr, perm, givptr*INTEGER. Arrays, dimension ($n \log_2 n$) each.

The array *prmptr*(*) contains a list of pointers which indicate where in *perm* a level's permutation is stored. $\text{prmptr}(i+1) - \text{prmptr}(i)$ indicates the size of the permutation and also the size of the full, non-deflated problem.

The array *perm*(*) contains the permutations (from deflation and sorting) to be applied to each eigenblock.

The array *givptr*(*) contains a list of pointers which indicate where in *givcol* a level's Givens rotations are stored. $\text{givptr}(i+1) - \text{givptr}(i)$ indicates the number of Givens rotations.

*givcol*INTEGER. Array, dimension (2, $n \log_2 n$).

Each pair of numbers indicates a pair of columns to take place in a Givens rotation.

*givnum*REAL for *slaeda*DOUBLE PRECISION for *dlaeda*.Array, dimension (2, $n \log_2 n$).

Each number indicates the *S* value to be used in the corresponding Givens rotation.

*q*REAL for *slaeda*DOUBLE PRECISION for *dlaeda*.Array, dimension (n^2).

Contains the square eigenblocks from previous levels, the starting positions for blocks are given by *qptr*.

qptr

INTEGER. Array, dimension ($n+2$). Contains a list of pointers which indicate where in *q* an eigenblock is stored. $\sqrt{\text{qptr}(i+1) - \text{qptr}(i)}$ indicates the size of the block.

*ztemp*REAL for *slaeda*DOUBLE PRECISION for *dlaeda*.Workspace array, dimension (n).

Output Parameters

*z*REAL for *slaeda*DOUBLE PRECISION for *dlaeda*.

Array, dimension (n). Contains the updating vector (the last row of the first sub-eigenvector matrix and the first row of the second sub-eigenvector matrix).

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = $-i$, the *i*-th parameter had an illegal value.

?laein

Computes a specified right or left eigenvector of an upper Hessenberg matrix by inverse iteration.

Syntax

```
call slaein( rightv, noinit, n, h, ldh, wr, wi, vr, vi, b, ldb, work, eps3, smlnum,
bignum, info )

call dlaein( rightv, noinit, n, h, ldh, wr, wi, vr, vi, b, ldb, work, eps3, smlnum,
bignum, info )

call claein( rightv, noinit, n, h, ldh, w, v, b, ldb, rwork, eps3, smlnum, info )

call zlaein( rightv, noinit, n, h, ldh, w, v, b, ldb, rwork, eps3, smlnum, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?laein uses inverse iteration to find a right or left eigenvector corresponding to the eigenvalue (wr,wi) of a real upper Hessenberg matrix H (for real flavors slaein/dlaein) or to the eigenvalue w of a complex upper Hessenberg matrix H (for complex flavors claein/zlaein).

Input Parameters

<i>rightv</i>	LOGICAL.
	If <i>rightv</i> = .TRUE., compute right eigenvector;
	if <i>rightv</i> = .FALSE., compute left eigenvector.
<i>noinit</i>	LOGICAL.
	If <i>noinit</i> = .TRUE., no initial vector is supplied in (vr,vi) or in v (for complex flavors);
	if <i>noinit</i> = .FALSE., initial vector is supplied in (vr,vi) or in v (for complex flavors).
<i>n</i>	INTEGER. The order of the matrix H ($n \geq 0$).
<i>h</i>	REAL for slaein DOUBLE PRECISION for dlaein COMPLEX for claein DOUBLE COMPLEX for zlaein. Array $h(ldh, *)$. The second dimension of h must be at least $\max(1, n)$. Contains the upper Hessenberg matrix H .
<i>ldh</i>	INTEGER. The leading dimension of the array h ; $ldh \geq \max(1, n)$.
<i>wr, wi</i>	REAL for slaein

DOUBLE PRECISION for `dlaein`.

The real and imaginary parts of the eigenvalue of H whose corresponding right or left eigenvector is to be computed (for real flavors of the routine).

w

COMPLEX for `claein`

DOUBLE COMPLEX for `zlaein`.

The eigenvalue of H whose corresponding right or left eigenvector is to be computed (for complex flavors of the routine).

vr, vi

REAL for `slaein`

DOUBLE PRECISION for `dlaein`.

Arrays, dimension (n) each. Used for real flavors only. On entry, if *noinit* = .FALSE. and *wi* = 0.0, *vr* must contain a real starting vector for inverse iteration using the real eigenvalue *wr*;

if *noinit* = .FALSE. and *wi* ≠ 0.0, *vr* and *vi* must contain the real and imaginary parts of a complex starting vector for inverse iteration using the complex eigenvalue (*wr,wi*); otherwise *vr* and *vi* need not be set.

v

COMPLEX for `claein`

DOUBLE COMPLEX for `zlaein`.

Array, dimension (n). Used for complex flavors only. On entry, if *noinit* = .FALSE., *v* must contain a starting vector for inverse iteration; otherwise *v* need not be set.

b

REAL for `slaein`

DOUBLE PRECISION for `dlaein`

COMPLEX for `claein`

DOUBLE COMPLEX for `zlaein`.

Workspace array *b*(*ldb, **). The second dimension of *b* must be at least $\max(1, n)$.

ldb

INTEGER. The leading dimension of the array *b*;

ldb ≥ *n+1* for real flavors;

ldb ≥ $\max(1, n)$ for complex flavors.

work

REAL for `slaein`

DOUBLE PRECISION for `dlaein`.

Workspace array, dimension (n).

Used for real flavors only.

rwork

REAL for `claein`

DOUBLE PRECISION for `zlaein`.

Workspace array, dimension (n).

Used for complex flavors only.

<i>eps3, smlnum</i>	REAL for slaein/claein DOUBLE PRECISION for dlaein/zlaein. <i>eps3</i> is a small machine-dependent value which is used to perturb close eigenvalues, and to replace zero pivots.
<i>smlnum</i>	A suggested value for <i>smlnum</i> is <i>slamch('s')</i> * (<i>n</i> / <i>slamch('p')</i>) for slaein/claein or <i>dlamch('s')</i> * (<i>n</i> / <i>dlamch('p')</i>) for dlaein/zlaein. See lamch .
<i>bignum</i>	REAL for slaein DOUBLE PRECISION for dlaein. <i>bignum</i> is a machine-dependent value close to overflow threshold. Used for real flavors only. A suggested value for <i>bignum</i> is $1 / \text{slamch('s')}$ for slaein/claein or $1 / \text{dlamch('s')}$ for dlaein/zlaein.

Output Parameters

<i>vr, vi</i>	On exit, if $w_i = 0.0$ (real eigenvalue), <i>vr</i> contains the computed real eigenvector; if $w_i \neq 0.0$ (complex eigenvalue), <i>vr</i> and <i>vi</i> contain the real and imaginary parts of the computed complex eigenvector. The eigenvector is normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be $ x + y $. <i>vi</i> is not referenced if $w_i = 0.0$.
<i>v</i>	On exit, <i>v</i> contains the computed eigenvector, normalized so that the component of largest magnitude has magnitude 1; here the magnitude of a complex number (x,y) is taken to be $ x + y $.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = 1, inverse iteration did not converge. For real flavors, <i>vr</i> is set to the last iterate, and so is <i>vi</i> , if $w_i \neq 0.0$. For complex flavors, <i>v</i> is set to the last iterate.

?laev2

Computes the eigenvalues and eigenvectors of a 2-by-2 symmetric/Hermitian matrix.

Syntax

```
call slaev2( a, b, c, rt1, rt2, cs1, sn1 )
call dlaev2( a, b, c, rt1, rt2, cs1, sn1 )
call claev2( a, b, c, rt1, rt2, cs1, sn1 )
call zlaev2( a, b, c, rt1, rt2, cs1, sn1 )
```

Include Files

- Fortran: mkl.fi

- C: mkl.h

Description

The routine performs the eigendecomposition of a 2-by-2 symmetric matrix

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix} \text{ (for } \text{slaev2/dlaev2) or Hermitian matrix } \begin{bmatrix} a & b \\ \text{conjg}(b) & c \end{bmatrix}$$

(for claev2/zlaev2).

On return, $rt1$ is the eigenvalue of larger absolute value, $rt2$ of smaller absolute value, and $(cs1, sn1)$ is the unit right eigenvector for $rt1$, giving the decomposition

$$\begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ b & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & sn1 \\ -sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for slaev2/dlaev2),

or

$$\begin{bmatrix} cs1 & \text{conjg}(sn1) \\ -sn1 & cs1 \end{bmatrix} \cdot \begin{bmatrix} a & b \\ \text{conjg}(b) & c \end{bmatrix} \cdot \begin{bmatrix} cs1 & -\text{conjg}(sn1) \\ sn1 & cs1 \end{bmatrix} = \begin{bmatrix} rt1 & 0 \\ 0 & rt2 \end{bmatrix}$$

(for claev2/zlaev2).

Input Parameters

a, b, c

- REAL for slaev2
- DOUBLE PRECISION for dlaev2
- COMPLEX for claev2
- DOUBLE COMPLEX for zlaev2.
- Elements of the input matrix.

Output Parameters

$rt1, rt2$

- REAL for slaev2/claeve2
- DOUBLE PRECISION for dlaev2/zlaev2.
- Eigenvalues of larger and smaller absolute value, respectively.

$cs1$

- REAL for slaev2/claeve2
- DOUBLE PRECISION for dlaev2/zlaev2.

$sn1$

- REAL for slaev2
- DOUBLE PRECISION for dlaev2

COMPLEX for `claev2`
 DOUBLE COMPLEX for `zlaev2`.

The vector ($cs1, sn1$) is the unit right eigenvector for $rt1$.

Application Notes

$rt1$ is accurate to a few ulps barring over/underflow. $rt2$ may be inaccurate if there is massive cancellation in the determinant $a*c - b*b$; higher precision or correctly rounded or correctly truncated arithmetic would be needed to compute $rt2$ accurately in all cases. $cs1$ and $sn1$ are accurate to a few ulps barring over/underflow. Overflow is possible only if $rt1$ is within a factor of 5 of overflow. Underflow is harmless if the input data is 0 or exceeds `underflow_threshold / macheps`.

?laexc

Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.

Syntax

```
call slaexc( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
call dlaexc( wantq, n, t, ldt, q, ldq, j1, n1, n2, work, info )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine swaps adjacent diagonal blocks T_{11} and T_{22} of order 1 or 2 in an upper quasi-triangular matrix T by an orthogonal similarity transformation.

T must be in Schur canonical form, that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

<i>wantq</i>	LOGICAL. If <i>wantq</i> = .TRUE., accumulate the transformation in the matrix Q ; If <i>wantq</i> = .FALSE., do not accumulate the transformation.
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>t, q</i>	REAL for <code>slaexc</code> DOUBLE PRECISION for <code>dlaexc</code> Arrays: $t(ldt, *)$ contains on entry the upper quasi-triangular matrix T , in Schur canonical form. The second dimension of t must be at least $\max(1, n)$.

q(1:dg,)* contains on entry, if *wantq* = .TRUE., the orthogonal matrix *Q*. If *wantq* = .FALSE., *q* is not referenced. The second dimension of *q* must be at least $\max(1, n)$.

ldt INTEGER. The leading dimension of *t*; at least $\max(1, n)$.

ldq INTEGER. The leading dimension of *q*;

If *wantq* = .FALSE., then *ldq* ≥ 1 .

If *wantq* = .TRUE., then *ldq* $\geq \max(1, n)$.

j1 INTEGER. The index of the first row of the first block T_{11} .

n1 INTEGER. The order of the first block T_{11}

(*n1* = 0, 1, or 2).

n2 INTEGER. The order of the second block T_{22}

(*n2* = 0, 1, or 2).

work REAL for *slaexc*;

DOUBLE PRECISION for *dlaexc*.

Workspace array, DIMENSION (*n*).

Output Parameters

t On exit, the updated matrix *T*, again in Schur canonical form.

q On exit, if *wantq* = .TRUE., the updated matrix *Q*.

info INTEGER.

If *info* = 0, the execution is successful.

If *info* = 1, the transformed matrix *T* would be too far from Schur form; the blocks are not swapped and *T* and *Q* are unchanged.

?lag2

Computes the eigenvalues of a 2-by-2 generalized eigenvalue problem, with scaling as necessary to avoid over-/underflow.

Syntax

```
call slag2( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
call dlag2( a, lda, b, ldb, safmin, scale1, scale2, wr1, wr2, wi )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the eigenvalues of a 2×2 generalized eigenvalue problem $A - w * B$, with scaling as necessary to avoid over-/underflow. The scaling factor, s , results in a modified eigenvalue equation

$$s^*A = w^*B,$$

where s is a non-negative scaling factor chosen so that w , w^*B , and s^*A do not overflow and, if possible, do not underflow, either.

Input Parameters

a, b	REAL for slag2 DOUBLE PRECISION for dlag2
Arrays:	
$a(1da,2)$	contains, on entry, the 2×2 matrix A . It is assumed that its 1-norm is less than $1/safmin$. Entries less than $\text{sqrt}(safmin) * \text{norm}(A)$ are subject to being treated as zero.
$b(ldb,2)$	contains, on entry, the 2×2 upper triangular matrix B . It is assumed that the one-norm of B is less than $1/safmin$. The diagonals should be at least $\text{sqrt}(safmin)$ times the largest element of B (in absolute value); if a diagonal is smaller than that, then $+/- \text{sqrt}(safmin)$ will be used instead of that diagonal.
$1da$	INTEGER. The leading dimension of a ; $1da \geq 2$.
$1db$	INTEGER. The leading dimension of b ; $1db \geq 2$.
$safmin$	REAL for slag2; DOUBLE PRECISION for dlag2. The smallest positive number such that $1/safmin$ does not overflow. (This should always be $?lamch('S')$ - it is an argument in order to avoid having to call $?lamch$ frequently.)

Output Parameters

$scale1$	REAL for slag2; DOUBLE PRECISION for dlag2.
	A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the first eigenvalue. If the eigenvalues are complex, then the eigenvalues are $(wr1 +/- wii)/scale1$ (which may lie outside the exponent range of the machine), $scale1=scale2$, and $scale1$ will always be positive. If the eigenvalues are real, then the first (real) eigenvalue is $wr1/scale1$, but this may overflow or underflow, and in fact, $scale1$ may be zero or less than the underflow threshhold if the exact eigenvalue is sufficiently large.
$scale2$	REAL for slag2; DOUBLE PRECISION for dlag2.
	A scaling factor used to avoid over-/underflow in the eigenvalue equation which defines the second eigenvalue. If the eigenvalues are complex, then $scale2=scale1$. If the eigenvalues are real, then the second (real)

eigenvalue is $wr2/scale2$, but this may overflow or underflow, and in fact, $scale2$ may be zero or less than the underflow threshold if the exact eigenvalue is sufficiently large.

wr1 REAL for slag2;

DOUBLE PRECISION for dlag2.

If the eigenvalue is real, then *wr1* is $scale1$ times the eigenvalue closest to the (2,2) element of $A^*inv(B)$.

If the eigenvalue is complex, then *wr1=wr2* is $scale1$ times the real part of the eigenvalues.

wr2 REAL for slag2;

DOUBLE PRECISION for dlag2.

If the eigenvalue is real, then *wr2* is $scale2$ times the other eigenvalue. If the eigenvalue is complex, then *wr1=wr2* is $scale1$ times the real part of the eigenvalues.

wi REAL for slag2;

DOUBLE PRECISION for dlag2.

If the eigenvalue is real, then *wi* is zero. If the eigenvalue is complex, then *wi* is $scale1$ times the imaginary part of the eigenvalues. *wi* will always be non-negative.

?lags2

Computes 2-by-2 orthogonal matrices U , V , and Q , and applies them to matrices A and B such that the rows of the transformed A and B are parallel.

Syntax

```
call slags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
call dlags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
call clags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
call zlags2( upper, a1, a2, a3, b1, b2, b3, csu, snu, csv, snv, csq, snq)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

For real flavors, the routine computes 2-by-2 orthogonal matrices U , V and Q , such that if *upper* = .TRUE., then

! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !

and

$$V^T * B * Q = V^T * \begin{bmatrix} B_1 & B_2 \\ 0 & B_3 \end{bmatrix} * Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

or if *upper* = .FALSE., then

$$U^T * A * Q = U^T * \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix} * Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

and

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The rows of the transformed A and B are parallel, where

$$U = \begin{bmatrix} csu & snu \\ -snu & csu \end{bmatrix}, V = \begin{bmatrix} csv & snv \\ -snv & csv \end{bmatrix}, Q = \begin{bmatrix} csq & snq \\ -snq & csq \end{bmatrix}$$

Here Z^T denotes the transpose of Z .

For complex flavors, the routine computes 2-by-2 unitary matrices U , V and Q , such that if *upper* = .TRUE., then

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

and

$$V^H * B * Q = V^H * \begin{bmatrix} B_1 & B_2 \\ 0 & B_3 \end{bmatrix} * Q = \begin{bmatrix} x & 0 \\ x & x \end{bmatrix}$$

or if *upper* = .FALSE., then

$$U^H * A * Q = U^H * \begin{bmatrix} A_1 & 0 \\ A_2 & A_3 \end{bmatrix} * Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

and

$$V^H * B * Q = V^H * \begin{bmatrix} B_1 & 0 \\ B_2 & B_3 \end{bmatrix} * Q = \begin{bmatrix} x & x \\ 0 & x \end{bmatrix}$$

The rows of the transformed A and B are parallel, where

$$U = \begin{bmatrix} csu & snu \\ -snu^H & csu \end{bmatrix}, \quad V = \begin{bmatrix} csv & snv \\ -snv^H & csv \end{bmatrix}, \quad Q = \begin{bmatrix} csq & snq \\ -snq^H & csq \end{bmatrix}$$

Input Parameters

<i>upper</i>	LOGICAL.
	If <i>upper</i> = .TRUE., the input matrices A and B are upper triangular; If <i>upper</i> = .FALSE., the input matrices A and B are lower triangular.
<i>a1, a3</i>	REAL for slags2 and clags2 DOUBLE PRECISION for dlags2 and zlags2
<i>a2</i>	REAL for slags2 DOUBLE PRECISION for dlags2 COMPLEX for clags2 COMPLEX*16 for zlags2 On entry, <i>a1, a2</i> and <i>a3</i> are elements of the input 2-by-2 upper (lower) triangular matrix A .
<i>b1, b3</i>	REAL for slags2 and clags2 DOUBLE PRECISION for dlags2 and zlags2
<i>b2</i>	REAL for slags2 DOUBLE PRECISION for dlags2 COMPLEX for clags2 COMPLEX*16 for zlags2 On entry, <i>b1, b2</i> and <i>b3</i> are elements of the input 2-by-2 upper (lower) triangular matrix B .

Output Parameters

<i>csu</i>	REAL for slags2 and clags2 DOUBLE PRECISION for dlags2 and zlags2 Element of the desired orthogonal matrix U .
<i>snu</i>	REAL for slags2 DOUBLE PRECISION for dlags2 Element of the desired orthogonal matrix U . COMPLEX for clags2 COMPLEX*16 for zlags2
<i>csv</i>	REAL for slags2 and clags2 DOUBLE PRECISION for dlags2 and zlags2 Element of the desired orthogonal matrix V .
<i>snv</i>	REAL for slags2 DOUBLE PRECISION for dlags2 COMPLEX for clags2 COMPLEX*16 for zlags2 Element of the desired orthogonal matrix V .
<i>csq</i>	REAL for slags2 and clags2 DOUBLE PRECISION for dlags2 and zlags2 Element of the desired orthogonal matrix Q .
<i>sng</i>	REAL for slags2 DOUBLE PRECISION for dlags2 Element of the desired orthogonal matrix Q . COMPLEX for clags2 COMPLEX*16 for zlags2

?lagtf

Computes an LU factorization of a matrix $T - \lambda * I$, where T is a general tridiagonal matrix, and λ is a scalar, using partial pivoting with row interchanges.

Syntax

```
call slagtf( n, a, lambda, b, c, tol, d, in, info )
call dlagtf( n, a, lambda, b, c, tol, d, in, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine factorizes the matrix ($T - \lambda I$), where T is an n -by- n tridiagonal matrix and λ is a scalar, as

$$T - \lambda I = P^* L^* U,$$

where P is a permutation matrix, L is a unit lower tridiagonal matrix with at most one non-zero sub-diagonal elements per column and U is an upper triangular matrix with at most two non-zero super-diagonal elements per column. The factorization is obtained by Gaussian elimination with partial pivoting and implicit row scaling. The parameter λ is included in the routine so that ?lagtf may be used, in conjunction with ?lagts, to obtain eigenvectors of T by inverse iteration.

Input Parameters

n	INTEGER. The order of the matrix T ($n \geq 0$).
a, b, c	REAL for slagtf DOUBLE PRECISION for dlagtf Arrays, dimension $a(n), b(n-1), c(n-1)$: On entry, $a(*)$ must contain the diagonal elements of the matrix T . On entry, $b(*)$ must contain the $(n-1)$ super-diagonal elements of T . On entry, $c(*)$ must contain the $(n-1)$ sub-diagonal elements of T .
tol	REAL for slagtf DOUBLE PRECISION for dlagtf On entry, a relative tolerance used to indicate whether or not the matrix ($T - \lambda I$) is nearly singular. tol should normally be chose as approximately the largest relative error in the elements of T . For example, if the elements of T are correct to about 4 significant figures, then tol should be set to about 5×10^{-4} . If tol is supplied as less than eps , where eps is the relative machine precision, then the value eps is used in place of tol .

Output Parameters

a	On exit, a is overwritten by the n diagonal elements of the upper triangular matrix U of the factorization of T .
b	On exit, b is overwritten by the $n-1$ super-diagonal elements of the matrix U of the factorization of T .
c	On exit, c is overwritten by the $n-1$ sub-diagonal elements of the matrix L of the factorization of T .
d	REAL for slagtf DOUBLE PRECISION for dlagtf Array, dimension $(n-2)$. On exit, d is overwritten by the $n-2$ second super-diagonal elements of the matrix U of the factorization of T .
in	INTEGER.

Array, dimension (n).

On exit, in contains details of the permutation matrix p . If an interchange occurred at the k -th step of the elimination, then $in(k) = 1$, otherwise $in(k) = 0$. The element $in(n)$ returns the smallest positive integer j such that

$$\text{abs}(u(j,j)) \leq \text{norm}((T - \lambda I)(j)) * \text{tol},$$

where $\text{norm}(A(j))$ denotes the sum of the absolute values of the j -th row of the matrix A .

If no such j exists then $in(n)$ is returned as zero. If $in(n)$ is returned as positive, then a diagonal element of U is small, indicating that $(T - \lambda I)$ is singular or nearly singular.

$info$

INTEGER.

If $info = 0$, the execution is successful.

If $info = -k$, the k -th parameter had an illegal value.

?lagtm

Performs a matrix-matrix product of the form $C = \alpha A^*B + \beta C$, where A is a tridiagonal matrix, B and C are rectangular matrices, and α and β are scalars, which may be 0, 1, or -1.

Syntax

```
call slagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call dlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call clagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
call zlagtm( trans, n, nrhs, alpha, dl, d, du, x, ldx, beta, b, ldb )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine performs a matrix-vector product of the form:

$$B := \alpha A^*X + \beta B$$

where A is a tridiagonal matrix of order n , B and X are n -by- $nrhs$ matrices, and α and β are real scalars, each of which may be 0., 1., or -1.

Input Parameters

$trans$

CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If $trans = 'N'$, then $B := \alpha A^*X + \beta B$ (no transpose);

If $trans = 'T'$, then $B := \alpha A^T X + \beta B$ (transpose);

If $\text{trans} = \text{'C'}$, then $B := \alpha A^H X + \beta B$ (conjugate transpose)

n INTEGER. The order of the matrix A ($n \geq 0$).

$nrhs$ INTEGER. The number of right-hand sides, i.e., the number of columns in X and B ($nrhs \geq 0$).

α, β REAL for `slagtm/clagtm`

DOUBLE PRECISION for `dlagtm/zlagtm`

Specify the scalars α and β respectively. α must be 0., 1., or -1.; otherwise, it is assumed to be 0. β must be 0., 1., or -1.; otherwise, it is assumed to be 1.

d_l, d, du REAL for `slagtm`

DOUBLE PRECISION for `dlagtm`

COMPLEX for `clagtm`

DOUBLE COMPLEX for `zlagtm`.

Arrays: $d_l(n - 1), d(n), du(n - 1)$.

The array d_l contains the $(n - 1)$ sub-diagonal elements of T .

The array d contains the n diagonal elements of T .

The array du contains the $(n - 1)$ super-diagonal elements of T .

x, b REAL for `slagtm`

DOUBLE PRECISION for `dlagtm`

COMPLEX for `clagtm`

DOUBLE COMPLEX for `zlagtm`.

Arrays:

$x(1:idx, :)$ contains the n -by- $nrhs$ matrix X . The second dimension of x must be at least $\max(1, nrhs)$.

$b(1:db, :)$ contains the n -by- $nrhs$ matrix B . The second dimension of b must be at least $\max(1, nrhs)$.

idx INTEGER. The leading dimension of the array x ; $idx \geq \max(1, n)$.

ldb INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

Output Parameters

b Overwritten by the matrix expression $B := \alpha A^H X + \beta B$

?lagts

Solves the system of equations $(T - \lambda I)^T x = y$ or $(T - \lambda I)x = y$, where T is a general tridiagonal matrix and λ is a scalar, using the LU factorization computed by ?lagtf.

Syntax

```
call slagts( job, n, a, b, c, d, in, y, tol, info )
call dlagts( job, n, a, b, c, d, in, y, tol, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine may be used to solve for x one of the systems of equations:

$$(T - \lambda I)x = y \text{ or } (T - \lambda I)^T x = y,$$

where T is an n -by- n tridiagonal matrix, following the factorization of $(T - \lambda I)$ as

$$T - \lambda I = P^* L^* U,$$

computed by the routine [?lagtf](#).

The choice of equation to be solved is controlled by the argument *job*, and in each case there is an option to perturb zero or very small diagonal elements of U , this option being intended for use in applications such as inverse iteration.

Input Parameters

<i>job</i>	INTEGER. Specifies the job to be performed by ?lagts as follows:
= 1:	The equations $(T - \lambda I)x = y$ are to be solved, but diagonal elements of U are not to be perturbed.
= -1:	The equations $(T - \lambda I)x = y$ are to be solved and, if overflow would otherwise occur, the diagonal elements of U are to be perturbed. See argument <i>tol</i> below.
= 2:	The equations $(T - \lambda I)^T x = y$ are to be solved, but diagonal elements of U are not to be perturbed.
= -2:	The equations $(T - \lambda I)^T x = y$ are to be solved and, if overflow would otherwise occur, the diagonal elements of U are to be perturbed. See argument <i>tol</i> below.
<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>a</i> , <i>b</i> , <i>c</i> , <i>d</i>	REAL for slagts DOUBLE PRECISION for dlagts Arrays, dimension $a(n)$, $b(n-1)$, $c(n-1)$, $d(n-2)$: On entry, $a(*)$ must contain the diagonal elements of U as returned from ?lagtf . On entry, $b(*)$ must contain the first super-diagonal elements of U as returned from ?lagtf . On entry, $c(*)$ must contain the sub-diagonal elements of L as returned from ?lagtf . On entry, $d(*)$ must contain the second super-diagonal elements of U as returned from ?lagtf .

in INTEGER.
 Array, dimension (*n*).
 On entry, *in*(*) must contain details of the matrix *p* as returned from ?lagtf.

y REAL for slagts
 DOUBLE PRECISION for dlagts
 Array, dimension (*n*). On entry, the right hand side vector *y*.

tol REAL for slagtf
 DOUBLE PRECISION for dlagtf.
 On entry, with *job* < 0, *tol* should be the minimum perturbation to be made to very small diagonal elements of *U*. *tol* should normally be chosen as about *eps**norm(*U*), where *eps* is the relative machine precision, but if *tol* is supplied as non-positive, then it is reset to *eps**max(abs(*u*(*i*,*j*))). If *job* > 0 then *tol* is not referenced.

Output Parameters

y On exit, *y* is overwritten by the solution vector *x*.

tol On exit, *tol* is changed as described in *Input Parameters* section above, only if *tol* is non-positive on entry. Otherwise *tol* is unchanged.

info INTEGER.
 If *info* = 0, the execution is successful.
 If *info* = -*i*, the *i*-th parameter had an illegal value. If *info* = *i* > 0, overflow would occur when computing the *i*th element of the solution vector *x*. This can only occur when *job* is supplied as positive and either means that a diagonal element of *U* is very small, or that the elements of the right-hand side vector *y* are very large.

?lagv2

Computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (*A,B*) where *B* is upper triangular.

Syntax

```
call slagv2( a, lda, b, ldb, alphai, beta, csl, snl, csr, snr )
call dlagv2( a, lda, b, ldb, alphai, beta, csl, snl, csr, snr )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the Generalized Schur factorization of a real 2-by-2 matrix pencil (A, B) where B is upper triangular. The routine computes orthogonal (rotation) matrices given by csl, snl and csr, snr such that:

1) if the pencil (A, B) has two real eigenvalues (include 0/0 or 1/0 types), then

$$\begin{bmatrix} a_{11} & a_{12} \\ 0 & a_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

2) if the pencil (A, B) has a pair of complex conjugate eigenvalues, then

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & 0 \\ 0 & b_{22} \end{bmatrix} = \begin{bmatrix} csl & snl \\ -snl & csl \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{bmatrix} \begin{bmatrix} csr & -snr \\ snr & csr \end{bmatrix}$$

where $b_{11} \geq b_{22} > 0$.

Input Parameters

a, b	REAL for slagv2 DOUBLE PRECISION for dlagv2
	Arrays: $a(lda,2)$ contains the 2-by-2 matrix A ; $b(ldb,2)$ contains the upper triangular 2-by-2 matrix B .

lda	INTEGER. The leading dimension of the array a ; $lda \geq 2$.
-------	---

ldb	INTEGER. The leading dimension of the array b ; $ldb \geq 2$.
-------	---

Output Parameters

a	On exit, a is overwritten by the "A-part" of the generalized Schur form.
b	On exit, b is overwritten by the "B-part" of the generalized Schur form.
$alphar, alphai, beta$	REAL for slagv2

DOUBLE PRECISION for dlagv2.

Arrays, dimension (2) each.

$(\alpha_{phar}(k) + i * \alpha_{phai}(k)) / \beta(k)$ are the eigenvalues of the pencil (A, B) , $k=1, 2$ and $i = \sqrt{-1}$.

Note that $\beta(k)$ may be zero.

csl, snl

REAL for slagv2

DOUBLE PRECISION for dlagv2

The cosine and sine of the left rotation matrix, respectively.

csr, snr

REAL for slagv2

DOUBLE PRECISION for dlagv2

The cosine and sine of the right rotation matrix, respectively.

?lahqr

Computes the eigenvalues and Schur factorization of an upper Hessenberg matrix, using the double-shift/single-shift QR algorithm.

Syntax

```
call slahqr( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, info )
call dlahqr( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, info )
call clahqr( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info )
call zlahqr( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine is an auxiliary routine called by ?hseqr to update the eigenvalues and Schur decomposition already computed by ?hseqr, by dealing with the Hessenberg submatrix in rows and columns ilo to ihi .

Input Parameters

$wantt$

LOGICAL.

If $wantt = .TRUE..$, the full Schur form T is required;

If $wantt = .FALSE..$, eigenvalues only are required.

$wantz$

LOGICAL.

If $wantz = .TRUE..$, the matrix of Schur vectors Z is required;

If $wantz = .FALSE..$, Schur vectors are not required.

n

INTEGER. The order of the matrix H ($n \geq 0$).

ilo, ihi

INTEGER.

It is assumed that h is already upper quasi-triangular in rows and columns $ihi+1:n$, and that $h(ilo, ilo-1) = 0$ (unless $ilo = 1$). The routine ?lahqr works primarily with the Hessenberg submatrix in rows and columns ilo to ihi , but applies transformations to all of h if $wantt = .TRUE..$

Constraints:

$$1 \leq ilo \leq \max(1, ihi); \quad ihi \leq n.$$
h, z

REAL for slahqr

DOUBLE PRECISION for dlahqr

COMPLEX for clahqr

DOUBLE COMPLEX for zlahqr.

Arrays:

 $h(ldh, *)$ contains the upper Hessenberg matrix h .The second dimension of h must be at least $\max(1, n)$. $z(ldz, *)$ If $wantz = .TRUE..$, then, on entry, z must contain the current matrix z of transformations accumulated by ?hseqr.If $wantz = .FALSE..$, then z is not referenced. The second dimension of z must be at least $\max(1, n)$.*ldh*INTEGER. The leading dimension of h ; at least $\max(1, n)$.*ldz*INTEGER. The leading dimension of z ; at least $\max(1, n)$.*iloz, ihiz*INTEGER. Specify the rows of z to which transformations must be applied if $wantz = .TRUE..$

$$1 \leq iloz \leq ilo; \quad ihi \leq ihiz \leq n.$$

Output Parameters

*h*On exit, if $info = 0$ and $wantt = .TRUE..$, then,

- for slahqr/dlahqr, h is upper quasi-triangular in rows and columns $ilo:ihi$ with any 2-by-2 diagonal blocks in standard form.
- for clahqr/zlahqr, h is upper triangular in rows and columns $ilo:ihi$.

If $info = 0$ and $wantt = .FALSE..$, the contents of h are unspecified on exit.If $info$ is positive, see description of $info$ for the output state of h .*wr, wi*

REAL for slahqr

DOUBLE PRECISION for dlahqr

Arrays, DIMENSION at least $\max(1, n)$ each. Used with real flavors only.The real and imaginary parts, respectively, of the computed eigenvalues ilo to ihi are stored in the corresponding elements of wr and wi . If two

eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of wr and wi , say the i -th and $(i+1)$ -th, with $wi(i) > 0$ and $wi(i+1) < 0$.

If $wantt = .TRUE.$, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in h , with $wr(i) = h(i,i)$, and, if $h(i:i+1, i:i+1)$ is a 2-by-2 diagonal block, $wi(i) = \sqrt{h(i+1,i)*h(i,i+1)}$ and $wi(i+1) = -wi(i)$.

w

COMPLEX for `clahqr`

DOUBLE COMPLEX for `zlahqr`.

Array, DIMENSION at least $\max(1, n)$. Used with complex flavors only. The computed eigenvalues ilo to ih_i are stored in the corresponding elements of w .

If $wantt = .TRUE.$, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in h , with $w(i) = h(i,i)$.

z

If $wantz = .TRUE.$, then, on exit z has been updated; transformations are applied only to the submatrix $z(iloz:ihiz, ilo:ih_i)$.

info

INTEGER.

If $info = 0$, the execution is successful.

With $info > 0$,

- if $info = i$, `?lahqr` failed to compute all the eigenvalues ilo to ih_i in a total of 30 iterations per eigenvalue; elements $i+1:ih_i$ of wr and wi (for `slahqr/dlahqr`) or w (for `clahqr/zlahqr`) contain those eigenvalues which have been successfully computed.
- if $wantt$ is $.FALSE.$, then on exit the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns ilo through $info$ of the final output value of h .
- if $wantt$ is $.TRUE.$, then on exit
 $(\text{initial value of } h)*u = u*(\text{final value of } h)$, (*)
where u is an orthogonal matrix. The final value of h is upper Hessenberg and triangular in rows and columns $info+1$ through ih_i .
- if $wantz$ is $.TRUE.$, then on exit
 $(\text{final value of } z) = (\text{initial value of } z)*u$,
where u is an orthogonal matrix in (*) regardless of the value of $wantt$.

?lahrd

Reduces the first nb columns of a general rectangular matrix A so that elements below the k -th subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A .

Syntax

```
call slahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call dlahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call clahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

```
call zlahrd( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine reduces the first nb columns of a real/complex general n -by- $(n-k+1)$ matrix A so that elements below the k -th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation $Q^T * A * Q$ for real flavors, or $Q^H * A * Q$ for complex flavors. The routine returns the matrices V and T which determine Q as a block reflector $I - V^* T^* V^T$ (for real flavors) or $I - V^* T^* V^H$ (for complex flavors), and also the matrix $Y = A^* V^* T$.

The matrix Q is represented as products of nb elementary reflectors:

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v^* v^T \text{ for real flavors, or}$$

$$H(i) = I - \tau v^* v^H \text{ for complex flavors, or}$$

where τ is a real/complex scalar, and v is a real/complex vector.

This is an obsolete auxiliary routine. Please use the new routine ?lahr2 instead.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>k</i>	INTEGER. The offset for the reduction. Elements below the k -th subdiagonal in the first nb columns are reduced to zero.
<i>nb</i>	INTEGER. The number of columns to be reduced.
<i>a</i>	REAL for slahrd DOUBLE PRECISION for dlahrd COMPLEX for clahrd DOUBLE COMPLEX for zlahrd. Array $a(la, n-k+1)$ contains the n -by- $(n-k+1)$ general matrix A to be reduced.
<i>lda</i>	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
<i>ldt</i>	INTEGER. The leading dimension of the output array t ; must be at least $\max(1, nb)$.
<i>ldy</i>	INTEGER. The leading dimension of the output array y ; must be at least $\max(1, n)$.

Output Parameters

<i>a</i>	On exit, the elements on and above the k -th subdiagonal in the first nb columns are overwritten with the corresponding elements of the reduced matrix; the elements below the k -th subdiagonal, with the array <i>tau</i> , represent the matrix Q as a product of elementary reflectors. The other columns of <i>a</i> are unchanged. See <i>Application Notes</i> below.
<i>tau</i>	<p>REAL for <code>slahrd</code></p> <p>DOUBLE PRECISION for <code>dlahrd</code></p> <p>COMPLEX for <code>clahrd</code></p> <p>DOUBLE COMPLEX for <code>zlahrd</code>.</p> <p>Array, DIMENSION (nb).</p> <p>Contains scalar factors of the elementary reflectors.</p>
<i>t, y</i>	<p>REAL for <code>slahrd</code></p> <p>DOUBLE PRECISION for <code>dlahrd</code></p> <p>COMPLEX for <code>clahrd</code></p> <p>DOUBLE COMPLEX for <code>zlahrd</code>.</p> <p>Arrays, dimension $t(ldt, nb)$, $y(ldy, nb)$.</p> <p>The array <i>t</i> contains upper triangular matrix T.</p> <p>The array <i>y</i> contains the n-by-nb matrix Y.</p>

Application Notes

For the elementary reflector $H(i)$,

$v(1:i+k-1) = 0$, $v(i+k) = 1$; $v(i+k+1:n)$ is stored on exit in *a*($i+k+1:n$, *i*) and *tau* is stored in *tau*(*i*).

The elements of the vectors *v* together form the $(n-k+1)$ -by- nb matrix *V* which is needed, with *T* and *Y*, to apply the transformation to the unreduced part of the matrix, using an update of the form:

$A := (I - V^* T^* V^T) * (A - Y^* V^T)$ for real flavors, or

$A := (I - V^* T^* V^H) * (A - Y^* V^H)$ for complex flavors.

The contents of *A* on exit are illustrated by the following example with $n = 7$, $k = 3$ and $nb = 2$:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v_1 & h & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix A , h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(i)$.

See Also

?lahr2

?lahr2

Reduces the specified number of first columns of a general rectangular matrix A so that elements below the specified subdiagonal are zero, and returns auxiliary matrices which are needed to apply the transformation to the unreduced part of A.

Syntax

```
call slahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call dlahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call clahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
call zlahr2( n, k, nb, a, lda, tau, t, ldt, y, ldy )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine reduces the first nb columns of a real/complex general n -by- $(n-k+1)$ matrix A so that elements below the k -th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation $Q^T * A * Q$ for real flavors, or $Q^H * A * Q$ for complex flavors. The routine returns the matrices V and T which determine Q as a block reflector $I - V^* T^* V^T$ (for real flavors) or $I - V^* T^* V^H$ (for real flavors), and also the matrix $Y = A^* V^* T$.

The matrix Q is represented as products of nb elementary reflectors:

$$Q = H(1)^* H(2)^* \dots ^* H(nb)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v^* v^T \text{ for real flavors, or}$$

$$H(i) = I - \tau v^* v^* v^H \text{ for complex flavors}$$

where τ is a real/complex scalar, and v is a real/complex vector.

This is an auxiliary routine called by ?gehrd.

Input Parameters

n	INTEGER. The order of the matrix A ($n \geq 0$).
k	INTEGER. The offset for the reduction. Elements below the k -th subdiagonal in the first nb columns are reduced to zero ($k < n$).
nb	INTEGER. The number of columns to be reduced.
a	REAL for slahr2 DOUBLE PRECISION for dlahr2

COMPLEX for `clahr2`
DOUBLE COMPLEX for `zlahr2`.

Array, DIMENSION ($lda, n-k+1$) contains the n -by- $(n-k+1)$ general matrix A to be reduced.

`lda` INTEGER. The leading dimension of the array a ; $lda \geq \max(1, n)$.

`ldt` INTEGER. The leading dimension of the output array t ; $ldt \geq nb$.

`ldy` INTEGER. The leading dimension of the output array y ; $ldy \geq n$.

Output Parameters

`a` On exit, the elements on and above the k -th subdiagonal in the first nb columns are overwritten with the corresponding elements of the reduced matrix; the elements below the k -th subdiagonal, with the array `tau`, represent the matrix Q as a product of elementary reflectors. The other columns of a are unchanged. See *Application Notes* below.

`tau` REAL for `slahr2`
DOUBLE PRECISION for `dlahr2`
COMPLEX for `clahr2`
DOUBLE COMPLEX for `zlahr2`.
Array, DIMENSION (nb).
Contains scalar factors of the elementary reflectors.

`t, y` REAL for `slahr2`
DOUBLE PRECISION for `dlahr2`
COMPLEX for `clahr2`
DOUBLE COMPLEX for `zlahr2`.
Arrays, dimension $t(ldt, nb), y(ldy, nb)$.
The array t contains upper triangular matrix T .
The array y contains the n -by- nb matrix Y .

Application Notes

For the elementary reflector $H(i)$,

$v(1:i+k-1) = 0, v(i+k) = 1; v(i+k+1:n)$ is stored on exit in $a(i+k+1:n, i)$ and `tau` is stored in `tau(i)`.

The elements of the vectors v together form the $(n-k+1)$ -by- nb matrix V which is needed, with T and Y , to apply the transformation to the unreduced part of the matrix, using an update of the form:

$A := (I - V^* T^* V^T) * (A - Y^* V^T)$ for real flavors, or

$A := (I - V^* T^* V^H) * (A - Y^* V^H)$ for complex flavors.

The contents of A on exit are illustrated by the following example with $n = 7, k = 3$ and $nb = 2$:

$$\begin{bmatrix} a & a & a & a & a \\ a & a & a & a & a \\ a & a & a & a & a \\ h & h & a & a & a \\ v_1 & h & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix A , h denotes a modified element of the upper Hessenberg matrix H , and v_i denotes an element of the vector defining $H(i)$.

?laic1

Applies one step of incremental condition estimation.

Syntax

```
call slaic1( job, j, x, sest, w, gamma, sestpr, s, c )
call dlaic1( job, j, x, sest, w, gamma, sestpr, s, c )
call claic1( job, j, x, sest, w, gamma, sestpr, s, c )
call zlaic1( job, j, x, sest, w, gamma, sestpr, s, c )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?laic1 applies one step of incremental condition estimation in its simplest version.

Let x , $\|x\|_2 = 1$ (where $\|a\|_2$ denotes the 2-norm of a), be an approximate singular vector of an j -by- j lower triangular matrix L , such that

$$\|L^*x\|_2 = sest$$

Then ?laic1 computes $sestpr, s, c$ such that the vector

$$\hat{x} = \begin{bmatrix} s^*x \\ c \end{bmatrix}$$

is an approximate singular vector of

$$\hat{L} = \begin{bmatrix} L & 0 \\ w^T & \gamma \end{bmatrix} \text{ (for complex flavors), or}$$

$$\hat{L} = \begin{bmatrix} L & 0 \\ w^T & \gamma \end{bmatrix} \text{ (for real flavors), in the sense that}$$

$$\|\hat{L}^*\hat{x}\|_2 = sestpr.$$

Depending on job , an estimate for the largest or smallest singular value is computed.

For real flavors, $[s \ c]^T$ and $sestpr^2$ is an eigenpair of the system

$$\text{diag}(sest^*sest, 0) + [\alpha \ \gamma] * \begin{bmatrix} \alpha \\ \gamma \end{bmatrix}$$

where $\alpha = x^T w$.

For complex flavors, $[s \ c]^H$ and $sestpr^2$ is an eigenpair of the system

$$\text{diag}(sest^*sest, 0) + [\alpha \ \gamma] * \begin{bmatrix} \text{conj}(\alpha) \\ \text{conj}(\gamma) \end{bmatrix}$$

where $\alpha = x^H w$.

Input Parameters

<i>job</i>	INTEGER. If <i>job</i> =1, an estimate for the largest singular value is computed; If <i>job</i> =2, an estimate for the smallest singular value is computed;
<i>j</i>	INTEGER. Length of <i>x</i> and <i>w</i> .
<i>x, w</i>	REAL for <i>slaic1</i> DOUBLE PRECISION for <i>dlaic1</i> COMPLEX for <i>claic1</i> DOUBLE COMPLEX for <i>zlaic1</i> . Arrays, dimension (<i>j</i>) each. Contain vectors <i>x</i> and <i>w</i> , respectively.
<i>sest</i>	REAL for <i>slaic1/claic1</i> ; DOUBLE PRECISION for <i>dlaic1/zlaic1</i> . Estimated singular value of <i>j</i> -by- <i>j</i> matrix <i>L</i> .
<i>gamma</i>	REAL for <i>slaic1</i> DOUBLE PRECISION for <i>dlaic1</i> COMPLEX for <i>claic1</i> DOUBLE COMPLEX for <i>zlaic1</i> . The diagonal element <i>gamma</i> .

Output Parameters

<i>sestpr</i>	REAL for <i>slaic1/claic1</i> ; DOUBLE PRECISION for <i>dlaic1/zlaic1</i> . Estimated singular value of (<i>j</i> +1)-by-(<i>j</i> +1) matrix <i>Lhat</i> .
---------------	---

?ln2

Solves a 1-by-1 or 2-by-2 linear system of equations of the specified form.

Syntax

```

call slaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2, b, ldb, wr, wi, x, ldx, scale,
xnorm, info )

call dlaln2( ltrans, na, nw, smin, ca, a, lda, d1, d2, b, ldb, wr, wi, x, ldx, scale,
xnorm, info )

```

Include Files

- Fortran: mkl.fi
 - C: mkl.h

Description

The routine solves a system of the form

$$(ca^*A - w^*D)^*X = s^*B, \text{ or } (ca^*A^T - w^*D)^*X = s^*B$$

with possible scaling (s) and perturbation of A .

A is an na -by- na real matrix, *ca* is a real scalar, *D* is an na -by- na real diagonal matrix, *w* is a real or complex value, and *X* and *B* are na -by-1 matrices: real if *w* is real, complex if *w* is complex. The parameter *na* may be 1 or 2.

If w is complex, X and B are represented as na -by-2 matrices, the first column of each being the real part and the second being the imaginary part.

The routine computes the scaling factor s (≤ 1) so chosen that X can be computed without overflow. X is further scaled if necessary to assure that $\text{norm}(ca^*A - w^*D) * \text{norm}(X)$ is less than overflow.

If both singular values of $(ca^*A - w^*D)$ are less than $smin$, $smin*I$ (where I stands for identity) will be used instead of $(ca^*A - w^*D)$. If only one singular value is less than $smin$, one element of $(ca^*A - w^*D)$ will be perturbed enough to make the smallest singular value roughly $smin$.

If both singular values are at least $smin$, $(ca^*A - w^*D)$ will not be perturbed. In any case, the perturbation will be at most some small multiple of $\max(smin, ulp * \text{norm}(ca^*A - w^*D))$.

The singular values are computed by infinity-norm approximations, and thus will only be correct to a factor of 2 or so.

NOTE

All input quantities are assumed to be smaller than overflow by a reasonable factor (see *bignum*).

Input Parameters

<i>trans</i>	LOGICAL. If <i>trans</i> = .TRUE., <i>A</i> - transpose will be used. If <i>trans</i> = .FALSE., <i>A</i> will be used (not transposed.)
<i>na</i>	INTEGER. The size of the matrix <i>A</i> , possible values 1 or 2.
<i>nw</i>	INTEGER. This parameter must be 1 if <i>w</i> is real, and 2 if <i>w</i> is complex. Possible values 1 or 2.
<i>smin</i>	REAL for <i>slaln2</i> DOUBLE PRECISION for <i>dlaln2</i> . The desired lower bound on the singular values of <i>A</i> . This should be a safe distance away from underflow or overflow, for example, between (<i>underflow/machine_precision</i>) and (<i>machine_precision * overflow</i>). (See <i>bignum</i> and <i>ulp</i>).
<i>ca</i>	REAL for <i>slaln2</i> DOUBLE PRECISION for <i>dlaln2</i> . The coefficient by which <i>A</i> is multiplied.
<i>a</i>	REAL for <i>slaln2</i> DOUBLE PRECISION for <i>dlaln2</i> . Array, DIMENSION (<i>lda,na</i>). The <i>na</i> -by- <i>na</i> matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> . Must be at least <i>na</i> .
<i>d1, d2</i>	REAL for <i>slaln2</i> DOUBLE PRECISION for <i>dlaln2</i> . The (1,1) and (2,2) elements in the diagonal matrix <i>D</i> , respectively. <i>d2</i> is not used if <i>nw</i> = 1.
<i>b</i>	REAL for <i>slaln2</i> DOUBLE PRECISION for <i>dlaln2</i> . Array, DIMENSION (<i>ldb,nw</i>). The <i>na</i> -by- <i>nw</i> matrix <i>B</i> (right-hand side). If <i>nw</i> = 2 (<i>w</i> is complex), column 1 contains the real part of <i>B</i> and column 2 contains the imaginary part.
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> . Must be at least <i>na</i> .
<i>wr, wi</i>	REAL for <i>slaln2</i> DOUBLE PRECISION for <i>dlaln2</i> . The real and imaginary part of the scalar <i>w</i> , respectively. <i>wi</i> is not used if <i>nw</i> = 1.

lidx INTEGER. The leading dimension of the output array *x*. Must be at least *na*.

Output Parameters

<i>x</i>	REAL for slaln2 DOUBLE PRECISION for dlaln2. Array, DIMENSION (<i>lidx,nw</i>). The <i>na</i> -by- <i>nw</i> matrix <i>X</i> (unknowns), as computed by the routine. If <i>nw</i> = 2 (<i>w</i> is complex), on exit, column 1 will contain the real part of <i>X</i> and column 2 will contain the imaginary part.
<i>scale</i>	REAL for slaln2 DOUBLE PRECISION for dlaln2. The scale factor that <i>B</i> must be multiplied by to insure that overflow does not occur when computing <i>X</i> . Thus $(ca^*A - w^*D) \cdot X$ will be <i>scale*B</i> , not <i>B</i> (ignoring perturbations of <i>A</i> .) It will be at most 1.
<i>xnorm</i>	REAL for slaln2 DOUBLE PRECISION for dlaln2. The infinity-norm of <i>X</i> , when <i>X</i> is regarded as an <i>na</i> -by- <i>nw</i> real matrix.
<i>info</i>	INTEGER. An error flag. It will be zero if no error occurs, a negative number if an argument is in error, or a positive number if $(ca^*A - w^*D)$ had to be perturbed. The possible values are: If <i>info</i> = 0: no error occurred, and $(ca^*A - w^*D)$ did not have to be perturbed. If <i>info</i> = 1: $(ca^*A - w^*D)$ had to be perturbed to make its smallest (or only) singular value greater than <i>smin</i> .

NOTE

For higher speed, this routine does not check the inputs for errors.

?lals0

Applies back multiplying factors in solving the least squares problem using divide and conquer SVD approach. Used by ?gelsd.

Syntax

```
call slals0( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol,
ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, info )

call dlals0( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol,
ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, info )

call clals0( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol,
ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, rwork, info )
```

```
call zlals0( icompq, nl, nr, sqre, nrhs, b, ldb, bx, ldbx, perm, givptr, givcol,
ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, rwork, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine applies back the multiplying factors of either the left or right singular vector matrix of a diagonal matrix appended by a row to the right hand side matrix B in solving the least squares problem using the divide-and-conquer SVD approach.

For the left singular vector matrix, three types of orthogonal matrices are involved:

(1L) Givens rotations: the number of such rotations is $givptr$; the pairs of columns/rows they were applied to are stored in $givcol$; and the c - and s -values of these rotations are stored in $givnum$.

(2L) Permutation. The $(n+1)$ -st row of B is to be moved to the first row, and for $j=2:n$, $perm(j)$ -th row of B is to be moved to the j -th row.

(3L) The left singular vector matrix of the remaining matrix.

For the right singular vector matrix, four types of orthogonal matrices are involved:

(1R) The right singular vector matrix of the remaining matrix.

(2R) If $sqre = 1$, one extra Givens rotation to generate the right null space.

(3R) The inverse transformation of (2L).

(4R) The inverse transformation of (1L).

Input Parameters

<i>icompq</i>	INTEGER. Specifies whether singular vectors are to be computed in factored form: If <i>icompq</i> = 0: Left singular vector matrix. If <i>icompq</i> = 1: Right singular vector matrix.
<i>nl</i>	INTEGER. The row dimension of the upper block. $nl \geq 1$.
<i>nr</i>	INTEGER. The row dimension of the lower block. $nr \geq 1$.
<i>sqre</i>	INTEGER. If <i>sqre</i> = 0: the lower block is an nr -by- nr square matrix. If <i>sqre</i> = 1: the lower block is an nr -by- $(nr+1)$ rectangular matrix. The bidiagonal matrix has row dimension $n = nl + nr + 1$, and column dimension $m = n + sqre$.
<i>nrhs</i>	INTEGER. The number of columns of B and bx . Must be at least 1.
<i>b</i>	REAL for slals0

DOUBLE PRECISION **for** `dlals0`
COMPLEX for `clals0`
DOUBLE COMPLEX for `zlals0.`
Array, DIMENSION (*ldb, nrhs*).

Contains the right hand sides of the least squares problem in rows 1 through *m*.

ldb
INTEGER. The leading dimension of *b*.
Must be at least $\max(1, \max(m, n))$.

bx
REAL for `slals0`
DOUBLE PRECISION for `dlals0`
COMPLEX for `clals0`
DOUBLE COMPLEX for `zlals0.`
Workspace array, DIMENSION (*ldbx, nrhs*).

ldbx
INTEGER. The leading dimension of *bx*.

perm
INTEGER. Array, DIMENSION (*n*).
The permutations (from deflation and sorting) applied to the two blocks.

givptr
INTEGER. The number of Givens rotations which took place in this subproblem.

givcol
INTEGER. Array, DIMENSION (*ldgcol, 2*). Each pair of numbers indicates a pair of rows/columns involved in a Givens rotation.

ldgcol
INTEGER. The leading dimension of *givcol*, must be at least *n*.

givnum
REAL for `slals0/clals0`
DOUBLE PRECISION for `dlals0/zlals0`
Array, DIMENSION (*ldgnum, 2*). Each number indicates the *c* or *s* value used in the corresponding Givens rotation.

ldgnum
INTEGER. The leading dimension of arrays *difr*, *poles* and *givnum*, must be at least *k*.

poles
REAL for `slals0/clals0`
DOUBLE PRECISION for `dlals0/zlals0`
Array, DIMENSION (*ldgnum, 2*). On entry, *poles(1:k, 1)* contains the new singular values obtained from solving the secular equation, and *poles(1:k, 2)* is an array containing the poles in the secular equation.

difl
REAL for `slals0/clals0`
DOUBLE PRECISION for `dlals0/zlals0`

Array, DIMENSION (k). On entry, $dif(i)$ is the distance between i -th updated (undeflated) singular value and the i -th (undeflated) old singular value.

difr

REAL for slals0/clals0

DOUBLE PRECISION for dlals0/zlals0

Array, DIMENSION ($ldgnum, 2$). On entry, $difr(i, 1)$ contains the distances between i -th updated (undeflated) singular value and the $i+1$ -th (undeflated) old singular value. And $difr(i, 2)$ is the normalizing factor for the i -th right singular vector.

z

REAL for slals0/clals0

DOUBLE PRECISION for dlals0/zlals0

Array, DIMENSION (k). Contains the components of the deflation-adjusted updating row vector.

K

INTEGER. Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. $1 \leq k \leq n$.

c

REAL for slals0/clals0

DOUBLE PRECISION for dlals0/zlals0

Contains garbage if $sqre = 0$ and the c value of a Givens rotation related to the right null space if $sqre = 1$.

s

REAL for slals0/clals0

DOUBLE PRECISION for dlals0/zlals0

Contains garbage if $sqre = 0$ and the s value of a Givens rotation related to the right null space if $sqre = 1$.

work

REAL for slals0

DOUBLE PRECISION for dlals0

Workspace array, DIMENSION (k). Used with real flavors only.

rwork

REAL for clals0

DOUBLE PRECISION for zlals0

Workspace array, DIMENSION ($k*(1+rhs) + 2*rhs$). Used with complex flavors only.

Output Parameters

b

On exit, contains the solution X in rows 1 through n .

info

INTEGER.

If $info = 0$: successful exit.

If $info = -i < 0$, the i -th argument had an illegal value.

?lalsa

Computes the SVD of the coefficient matrix in compact form. Used by ?gelsd.

Syntax

```
call slalsa( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
call dlalsa( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
call clalsa( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, rwork, iwork, info )
call zlalsa( icompq, smlsiz, n, nrhs, b, ldb, bx, ldbx, u, ldu, vt, k, difl, difr, z,
poles, givptr, givcol, ldgcol, perm, givnum, c, s, rwork, iwork, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine is an intermediate step in solving the least squares problem by computing the SVD of the coefficient matrix in compact form. The singular vectors are computed as products of simple orthogonal matrices.

If $icompq = 0$, ?lalsa applies the inverse of the left singular vector matrix of an upper bidiagonal matrix to the right hand side; and if $icompq = 1$, the routine applies the right singular vector matrix to the right hand side. The singular vector matrices were generated in the compact form by ?lalsa.

Input Parameters

<i>icompq</i>	INTEGER. Specifies whether the left or the right singular vector matrix is involved. If <i>icompq</i> = 0: left singular vector matrix is used If <i>icompq</i> = 1: right singular vector matrix is used.
<i>smlsiz</i>	INTEGER. The maximum size of the subproblems at the bottom of the computation tree.
<i>n</i>	INTEGER. The row and column dimensions of the upper bidiagonal matrix.
<i>nrhs</i>	INTEGER. The number of columns of <i>b</i> and <i>bx</i> . Must be at least 1.
<i>b</i>	REAL for slalsa DOUBLE PRECISION for dlalsa COMPLEX for clalsa DOUBLE COMPLEX for zlalsa Array, DIMENSION (<i>ldb</i> , <i>nrhs</i>). Contains the right hand sides of the least squares problem in rows 1 through <i>m</i> .

<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> in the calling subprogram. Must be at least $\max(1, \max(m, n))$.
<i>ldbx</i>	INTEGER. The leading dimension of the output array <i>bx</i> .
<i>u</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu, smlsiz</i>) . On entry, <i>u</i> contains the left singular vector matrices of all subproblems at the bottom level.
<i>ldu</i>	INTEGER, $ldu \geq n$. The leading dimension of arrays <i>u, vt, difl, difr, poles, givnum</i> , and <i>z</i> .
<i>vt</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu, smlsiz +1</i>) . On entry, vt^T (for real flavors) or vt^H (for complex flavors) contains the right singular vector matrices of all subproblems at the bottom level.
<i>k</i>	INTEGER array, DIMENSION (<i>n</i>).
<i>difl</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu, nlvl</i>) , where $nlvl = \text{int}(\log_2(n / (smlsiz+1))) + 1$.
<i>difr</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu, 2*nlvl</i>) . On entry, <i>difl(*, i)</i> and <i>difr(*, 2<i>i</i>-1)</i> record distances between singular values on the <i>i</i> -th level and singular values on the (<i>i</i> -1)-th level, and <i>difr(*, 2<i>i</i>)</i> record the normalizing factors of the right singular vectors matrices of subproblems on <i>i</i> -th level.
<i>z</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu, nlvl</i>) . On entry, <i>z(1, i)</i> contains the components of the deflation- adjusted updating the row vector for subproblems on the <i>i</i> -th level.
<i>poles</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu, 2*nlvl</i>) . On entry, <i>poles(*, 2<i>i</i>-1: 2<i>i</i>)</i> contains the new and old singular values involved in the secular equations on the <i>i</i> -th level.
<i>givptr</i>	INTEGER. Array, DIMENSION (<i>n</i>). On entry, <i>givptr(i)</i> records the number of Givens rotations performed on the <i>i</i> -th problem on the computation tree.

<i>givcol</i>	INTEGER. Array, DIMENSION (<i>ldgcol</i> , $2*nvl$). On entry, for each i , <i>givcol</i> (*, $2i-1 : 2i$) records the locations of Givens rotations performed on the i -th level on the computation tree.
<i>ldgcol</i>	INTEGER, <i>ldgcol</i> $\geq n$. The leading dimension of arrays <i>givcol</i> and <i>perm</i> .
<i>perm</i>	INTEGER. Array, DIMENSION (<i>ldgcol</i> , nvl). On entry, <i>perm</i> (*, i) records permutations done on the i -th level of the computation tree.
<i>givnum</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (<i>ldu</i> , $2*nvl$). On entry, <i>givnum</i> (*, $2i-1 : 2i$) records the <i>c</i> and <i>s</i> values of Givens rotations performed on the i -th level on the computation tree.
<i>c</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (n). On entry, if the i -th subproblem is not square, <i>c</i> (i) contains the <i>c</i> value of a Givens rotation related to the right null space of the i -th subproblem.
<i>s</i>	REAL for slalsa/clalsa DOUBLE PRECISION for dlalsa/zlalsa Array, DIMENSION (n). On entry, if the i -th subproblem is not square, <i>s</i> (i) contains the <i>s</i> -value of a Givens rotation related to the right null space of the i -th subproblem.
<i>work</i>	REAL for slalsa DOUBLE PRECISION for dlalsa Workspace array, DIMENSION at least (n). Used with real flavors only.
<i>rwork</i>	REAL for clalsa DOUBLE PRECISION for zlalsa Workspace array, DIMENSION at least $\max(n, (smlsz+1)*nrhs*3)$. Used with complex flavors only.
<i>iwork</i>	INTEGER. Workspace array, DIMENSION at least ($3n$).

Output Parameters

<i>b</i>	On exit, contains the solution X in rows 1 through n .
<i>bx</i>	REAL for slalsa DOUBLE PRECISION for dlalsa COMPLEX for clalsa DOUBLE COMPLEX for zlalsa

Array, DIMENSION ($ldb, nrhs$). On exit, the result of applying the left or right singular vector matrix to b .

info INTEGER. If $info = 0$: successful exit

If $info = -i < 0$, the i -th argument had an illegal value.

?lalsd

Uses the singular value decomposition of A to solve the least squares problem.

Syntax

```
call slalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, iwork, info )
call dlalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, iwork, info )
call clalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, rwork, iwork,
info )
call zlalsd( uplo, smlsiz, n, nrhs, d, e, b, ldb, rcond, rank, work, rwork, iwork,
info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine uses the singular value decomposition of A to solve the least squares problem of finding X to minimize the Euclidean norm of each column of A^*X-B , where A is n -by- n upper bidiagonal, and X and B are n -by- $nrhs$. The solution X overwrites B .

The singular values of A smaller than $rcond$ times the largest singular value are treated as zero in solving the least squares problem; in this case a minimum norm solution is returned. The actual singular values are returned in d in ascending order.

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2.

It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

Input Parameters

<i>uplo</i>	CHARACTER*1. If $uplo = 'U'$, d and e define an upper bidiagonal matrix. If $uplo = 'L'$, d and e define a lower bidiagonal matrix.
<i>smlsiz</i>	INTEGER. The maximum size of the subproblems at the bottom of the computation tree.
<i>n</i>	INTEGER. The dimension of the bidiagonal matrix. $n \geq 0$.

<i>nrhs</i>	INTEGER. The number of columns of <i>B</i> . Must be at least 1.
<i>d</i>	<p>REAL for <i>slalsd/clalsd</i></p> <p>DOUBLE PRECISION for <i>dlalsd/zlalsd</i></p> <p>Array, DIMENSION (<i>n</i>). On entry, <i>d</i> contains the main diagonal of the bidiagonal matrix.</p>
<i>e</i>	<p>REAL for <i>slalsd/clalsd</i></p> <p>DOUBLE PRECISION for <i>dlalsd/zlalsd</i></p> <p>Array, DIMENSION (<i>n-1</i>). Contains the super-diagonal entries of the bidiagonal matrix. On exit, <i>e</i> is destroyed.</p>
<i>b</i>	<p>REAL for <i>slalsd</i></p> <p>DOUBLE PRECISION for <i>dlalsd</i></p> <p>COMPLEX for <i>clalsd</i></p> <p>DOUBLE COMPLEX for <i>zlalsd</i></p> <p>Array, DIMENSION (<i>ldb,nrhs</i>).</p> <p>On input, <i>b</i> contains the right hand sides of the least squares problem. On output, <i>b</i> contains the solution <i>X</i>.</p>
<i>ldb</i>	INTEGER. The leading dimension of <i>b</i> in the calling subprogram. Must be at least $\max(1, n)$.
<i>rcond</i>	<p>REAL for <i>slalsd/clalsd</i></p> <p>DOUBLE PRECISION for <i>dlalsd/zlalsd</i></p> <p>The singular values of <i>A</i> less than or equal to <i>rcond</i> times the largest singular value are treated as zero in solving the least squares problem. If <i>rcond</i> is negative, machine precision is used instead. For example, for the least squares problem <i>diag(S)*X=B</i>, where <i>diag(S)</i> is a diagonal matrix of singular values, the solution is $X(i)=B(i)/S(i)$ if <i>S(i)</i> is greater than <i>rcond *max(S)</i>, and $X(i)=0$ if <i>S(i)</i> is less than or equal to <i>rcond *max(S)</i>.</p>
<i>rank</i>	INTEGER. The number of singular values of <i>A</i> greater than <i>rcond</i> times the largest singular value.
<i>work</i>	<p>REAL for <i>slalsd</i></p> <p>DOUBLE PRECISION for <i>dlalsd</i></p> <p>COMPLEX for <i>clalsd</i></p> <p>DOUBLE COMPLEX for <i>zlalsd</i></p> <p>Workspace array.</p> <p>DIMENSION for real flavors at least</p> $(9n+2n*smlsiz+8n*nlvl+n*nrhs+(smlsiz+1)^2),$ <p>where</p> <p><i>nlvl</i> = $\max(0, \text{int}(\log_2(n/(smlsiz+1)))) + 1$.</p>

DIMENSION for complex flavors is $(n*nrhs)$.

rwork REAL for clalsd
 DOUBLE PRECISION for zlalsd
 Workspace array, used with complex flavors only.
 DIMENSION at least $(9n + 2n*smlsiz + 8n*nlvl + 3*mlsiz*nrhs + (smlsiz+1)^2)$,
 where
 $nlvl = \max(0, \text{int}(\log_2(\min(m,n)/(smlsiz+1))) + 1)$.
iwork INTEGER.
 Workspace array of DIMENSION $(3n*nlvl + 11n)$.

Output Parameters

d On exit, if $\text{info} = 0$, *d* contains singular values of the bidiagonal matrix.
e On exit, destroyed.
b On exit, *b* contains the solution X .
info INTEGER.
 If $\text{info} = 0$: successful exit.
 If $\text{info} = -i < 0$, the *i*-th argument had an illegal value.
 If $\text{info} > 0$: The algorithm failed to compute a singular value while working on the submatrix lying in rows and columns $\text{info}/(n+1)$ through $\text{mod}(\text{info}, n+1)$.

?lamrg

Creates a permutation list to merge the entries of two independently sorted sets into a single set sorted in ascending order.

Syntax

```
call slamrg( n1, n2, a, strd1, strd2, index )
call dlamrg( n1, n2, a, strd1, strd2, index )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine creates a permutation list which will merge the elements of *a* (which is composed of two independently sorted sets) into a single set which is sorted in ascending order.

Input Parameters

<i>n1, n2</i>	INTEGER. These arguments contain the respective lengths of the two sorted lists to be merged.
<i>a</i>	REAL for slamrg DOUBLE PRECISION for dlamrg. Array, DIMENSION (<i>n1+n2</i>). The first <i>n1</i> elements of <i>a</i> contain a list of numbers which are sorted in either ascending or descending order. Likewise for the final <i>n2</i> elements.
<i>strd1, strd2</i>	INTEGER. These are the strides to be taken through the array <i>a</i> . Allowable strides are 1 and -1. They indicate whether a subset of <i>a</i> is sorted in ascending (<i>strdx</i> = 1) or descending (<i>strdx</i> = -1) order.

Output Parameters

<i>index</i>	INTEGER. Array, DIMENSION (<i>n1+n2</i>). On exit, this array will contain a permutation such that if $b(i) = a(index(i))$ for $i=1, n1+n2$, then <i>b</i> will be sorted in ascending order.
--------------	---

?laneg

Computes the Sturm count, the number of negative pivots encountered while factoring tridiagonal T -
 $\sigma * I = L * D * L^T$.

Syntax

```
value = slaneg( n, d, lld, sigma, pivmin, r )
value = dlaneg( n, d, lld, sigma, pivmin, r )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the Sturm count, the number of negative pivots encountered while factoring tridiagonal T - $\sigma * I = L * D * L^T$. This implementation works directly on the factors without forming the tridiagonal matrix T . The Sturm count is also the number of eigenvalues of T less than σ . This routine is called from ?larb. The current routine does not use the *pivmin* parameter but rather requires IEEE-754 propagation of infinities and NaNs (NaN stands for 'Not A Number'). This routine also has no input range restrictions but does require default exception handling such that $x/0$ produces Inf when x is non-zero, and Inf/Inf produces NaN. (For more information see [Marques06]).

Input Parameters

<i>n</i>	INTEGER. The order of the matrix.
----------	-----------------------------------

<i>d</i>	REAL for slaneg DOUBLE PRECISION for dlaneg Array, DIMENSION (n). Contains n diagonal elements of the matrix D .
<i>lld</i>	REAL for slaneg DOUBLE PRECISION for dlaneg Array, DIMENSION (n-1). Contains $(n-1)$ elements $L(i) * L(i) * D(i)$.
<i>sigma</i>	REAL for slaneg DOUBLE PRECISION for dlaneg Shift amount in $T - \sigma I = L^* D^* L^{**} T$.
<i>pivmin</i>	REAL for slaneg DOUBLE PRECISION for dlaneg The minimum pivot in the Sturm sequence. May be used when zero pivots are encountered on non-IEEE-754 architectures.
<i>r</i>	INTEGER. The twist index for the twisted factorization that is used for the negcount.

Output Parameters

<i>value</i>	INTEGER. The number of negative pivots encountered while factoring.
--------------	---

?langb

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of general band matrix.

Syntax

```
val = slangb( norm, n, kl, ku, ab, ldab, work )
val = dlangb( norm, n, kl, ku, ab, ldab, work )
val = clangb( norm, n, kl, ku, ab, ldab, work )
val = zlangb( norm, n, kl, ku, ab, ldab, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n band matrix A , with kl sub-diagonals and ku super-diagonals.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': <i>val</i> = $\max(A_{ij})$, largest absolute value of the matrix <i>A</i> . = '1' or 'O' or 'o': <i>val</i> = $\text{norm1}(A)$, 1-norm of the matrix <i>A</i> (maximum column sum), = 'I' or 'i': <i>val</i> = $\text{normI}(A)$, infinity norm of the matrix <i>A</i> (maximum row sum), = 'F', 'f', 'E' or 'e': <i>val</i> = $\text{normF}(A)$, Frobenius norm of the matrix <i>A</i> (square root of sum of squares).
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$. When <i>n</i> = 0, ?langb is set to zero.
<i>kl</i>	INTEGER. The number of sub-diagonals of the matrix <i>A</i> . $kl \geq 0$.
<i>ku</i>	INTEGER. The number of super-diagonals of the matrix <i>A</i> . $ku \geq 0$.
<i>ab</i>	REAL for slangb DOUBLE PRECISION for dlangb COMPLEX for clangb DOUBLE COMPLEX for zlangb Array, DIMENSION (<i>ldab,n</i>). The band matrix <i>A</i> , stored in rows 1 to <i>kl+ku+1</i> . The <i>j</i> -th column of <i>A</i> is stored in the <i>j</i> -th column of the array <i>ab</i> as follows: $ab(ku+1+i-j, j) = a(i, j)$ $\text{for } \max(1, j-ku) \leq i \leq \min(n, j+kl).$
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> . $ldab \geq kl+ku+1$.
<i>work</i>	REAL for slangb/clangb DOUBLE PRECISION for dlangb/zlangb Workspace array, DIMENSION ($\max(1, lwork)$), where $lwork \geq n$ when <i>norm</i> = 'I'; otherwise, <i>work</i> is not referenced.

Output Parameters

<i>val</i>	REAL for slangb/clangb DOUBLE PRECISION for dlangb/zlangb Value returned by the function.
------------	---

?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.

Syntax

```
val = slange( norm, m, n, a, lda, work )
val = dlange( norm, m, n, a, lda, work )
val = clange( norm, m, n, a, lda, work )
val = zlange( norm, m, n, a, lda, work )
```

C:

```
float LAPACKE_slange (int matrix_layout, char norm, lapack_int m, lapack_int n, const
float * a, lapack_int lda);

double LAPACKE_dlange (int matrix_layout, char norm, lapack_int m, lapack_int n, const
double * a, lapack_int lda);

float LAPACKE_clange (int matrix_layout, char norm, lapack_int m, lapack_int n, const
lapack_complex_float * a, lapack_int lda);

double LAPACKE_zlange (int matrix_layout, char norm, lapack_int m, lapack_int n, const
lapack_complex_double * a, lapack_int lda);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lange returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex matrix A.

Input Parameters

The data types are given for the Fortran interface.

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': val = max(abs(A _{ij})), largest absolute value of the matrix A. = '1' or 'O' or 'o': val = norm1(A), 1-norm of the matrix A (maximum column sum), = 'I' or 'i': val = normI(A), infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': val = normF(A), Frobenius norm of the matrix A (square root of sum of squares).
<i>m</i>	INTEGER. The number of rows of the matrix A. <i>m</i> ≥ 0. When <i>m</i> = 0, ?lange is set to zero.
<i>n</i>	INTEGER. The number of columns of the matrix A.

$n \geq 0$. When $n = 0$, ?lange is set to zero.

a REAL for slange

DOUBLE PRECISION for dlange

COMPLEX for clange

DOUBLE COMPLEX for zlange

Array, DIMENSION (*lda,n*).

The *m*-by-*n* matrix *A*.

lda INTEGER. The leading dimension of the array *a*.

lda $\geq \max(m, 1)$.

work REAL for slange and clange.

DOUBLE PRECISION for dlange and zlange.

Workspace array, DIMENSION $\max(1, lwork)$, where $lwork \geq m$ when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val REAL for slange/clange

DOUBLE PRECISION for dlange/zlange

Value returned by the function.

?langt

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general tridiagonal matrix.

Syntax

val = slangt(*norm*, *n*, *dl*, *d*, *du*)

val = dlangt(*norm*, *n*, *dl*, *d*, *du*)

val = clangt(*norm*, *n*, *dl*, *d*, *du*)

val = zlangt(*norm*, *n*, *dl*, *d*, *du*)

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex tridiagonal matrix *A*.

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': *val* = max(abs(A_{ij})), largest absolute value of the matrix *A*.
= '1' or 'O' or 'o': *val* = norm1(*A*), 1-norm of the matrix *A* (maximum column sum),
= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),
= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

n INTEGER. The order of the matrix *A*. $n \geq 0$. When *n* = 0, ?langt is set to zero.

dl, d, du

```
REAL for slangt
DOUBLE PRECISION for dlangt
COMPLEX for clangt
DOUBLE COMPLEX for zlangt
```

Arrays: *dl* (*n*-1), *d* (*n*), *du* (*n*-1).
The array *dl* contains the (*n*-1) sub-diagonal elements of *A*.
The array *d* contains the diagonal elements of *A*.
The array *du* contains the (*n*-1) super-diagonal elements of *A*.

Output Parameters

val

```
REAL for slangt/clangt
DOUBLE PRECISION for dlangt/zlangt
```

Value returned by the function.

?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of an upper Hessenberg matrix.

Syntax

```
val = slanh( norm, n, a, lda, work )
val = dlanh( norm, n, a, lda, work )
val = clanh( norm, n, a, lda, work )
val = zlanh( norm, n, a, lda, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lanhs returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hessenberg matrix *A*.

The value *val* returned by the function is:

```

val = max(abs(Aij)), if norm = 'M' or 'm'
= norm1(A), if norm = '1' or 'O' or 'o'
= normI(A), if norm = 'I' or 'i'
= normF(A), if norm = 'F', 'f', 'E' or 'e'
```

where `norm1` denotes the 1-norm of a matrix (maximum column sum), `normI` denotes the infinity norm of a matrix (maximum row sum) and `normF` denotes the Frobenius norm of a matrix (square root of sum of squares). Note that `max(abs(Aij))` is not a consistent matrix norm.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine as described above.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$. When <i>n</i> = 0, ?lanhs is set to zero.
<i>a</i>	REAL for slanh DOUBLE PRECISION for dlanhs COMPLEX for clanhs DOUBLE COMPLEX for zlanhs Array, DIMENSION (<i>lda,n</i>). The <i>n</i> -by- <i>n</i> upper Hessenberg matrix <i>A</i> ; the part of <i>A</i> below the first sub-diagonal is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . <i>lda</i> $\geq \max(n, 1)$.
<i>work</i>	REAL for slanh and clanhs. DOUBLE PRECISION for dlanhs and zlanhs. Workspace array, DIMENSION ($\max(1, lwork)$), where <i>lwork</i> $\geq n$ when <i>norm</i> = 'I'; otherwise, <i>work</i> is not referenced.

Output Parameters

<i>val</i>	REAL for slanh/clanh DOUBLE PRECISION for dlanhs/zlanhs Value returned by the function.
------------	---

?lanhs

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric band matrix.

Syntax

```

val = slansb( norm, uplo, n, k, ab, ldab, work )
val = dlansb( norm, uplo, n, k, ab, ldab, work )
val = clansb( norm, uplo, n, k, ab, ldab, work )
```

```
val = zlansb( norm, uplo, n, k, ab, ldab, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lansb returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n real/complex symmetric band matrix A , with k super-diagonals.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine:
= 'M' or 'm':	<i>val</i> = $\max(A_{ij})$, largest absolute value of the matrix A .
= '1' or 'O' or 'o':	<i>val</i> = $\text{norm}_1(A)$, 1-norm of the matrix A (maximum column sum),
= 'I' or 'i':	<i>val</i> = $\text{norm}_I(A)$, infinity norm of the matrix A (maximum row sum),
= 'F', 'f', 'E' or 'e':	<i>val</i> = $\text{norm}_F(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix A is supplied. If <i>uplo</i> = 'U': upper triangular part is supplied; If <i>uplo</i> = 'L': lower triangular part is supplied.
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, ?lansb is set to zero.
<i>k</i>	INTEGER. The number of super-diagonals or sub-diagonals of the band matrix A . $k \geq 0$.
<i>ab</i>	REAL for slansb DOUBLE PRECISION for dlansb COMPLEX for clansb DOUBLE COMPLEX for zlansb Array, DIMENSION (<i>ldab,n</i>). The upper or lower triangle of the symmetric band matrix A , stored in the first $k+1$ rows of <i>ab</i> . The j -th column of A is stored in the j -th column of the array <i>ab</i> as follows: if <i>uplo</i> = 'U', <i>ab</i> ($k+1+i-j,j$) = <i>a</i> (i,j) for $\max(1,j-k) \leq i \leq j$; if <i>uplo</i> = 'L', <i>ab</i> ($1+i-j,j$) = <i>a</i> (i,j) for $j \leq i \leq \min(n, j+k)$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> .

$ldab \geq k+1$.

work

REAL for slansb and clansb.

DOUBLE PRECISION for dlansb and zlansb.

Workspace array, DIMENSION ($\max(1, lwork)$), where

$lwork \geq n$ when $norm = 'I'$ or ' 1 ' or ' O '; otherwise, *work* is not referenced.

Output Parameters

val

REAL for slansb/clansb

DOUBLE PRECISION for dlansb/zlansb

Value returned by the function.

?lanhb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian band matrix.

Syntax

```
val = clanhb( norm, uplo, n, k, ab, ldab, work )
```

```
val = zlanhb( norm, uplo, n, k, ab, ldab, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n Hermitian band matrix A , with k super-diagonals.

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

- = ' M ' or ' m ': *val* = $\max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .
- = ' 1 ' or ' O ' or ' o ': *val* = $\text{norm}_1(A)$, 1-norm of the matrix A (maximum column sum),
- = ' I ' or ' i ': *val* = $\text{norm}_I(A)$, infinity norm of the matrix A (maximum row sum),
- = ' F ', ' f ', ' E ' or ' e ': *val* = $\text{norm}_F(A)$, Frobenius norm of the matrix A (square root of sum of squares).

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the band matrix A is supplied.

If *uplo* = ' U ': upper triangular part is supplied;

If $uplo = 'L'$: lower triangular part is supplied.

n INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, ?lanhb is set to zero.

k INTEGER. The number of super-diagonals or sub-diagonals of the band matrix A .

$k \geq 0$.

ab COMPLEX for clanhb.

DOUBLE COMPLEX for zlanhb.

Array, DIMENSION (l_{daB}, n). The upper or lower triangle of the Hermitian band matrix A , stored in the first $k+1$ rows of ab . The j -th column of A is stored in the j -th column of the array ab as follows:

```
if  $uplo = 'U'$ ,  $ab(k+1+i-j, j) = a(i, j)$ 
for  $\max(1, j-k) \leq i \leq j$ ;
if  $uplo = 'L'$ ,  $ab(1+i-j, j) = a(i, j)$  for  $j \leq i \leq \min(n, j+k)$ .
```

Note that the imaginary parts of the diagonal elements need not be set and are assumed to be zero.

l_{dab} INTEGER. The leading dimension of the array ab . $l_{dab} \geq k+1$.

$work$ REAL for clanhb.

DOUBLE PRECISION for zlanhb.

Workspace array, DIMENSION $\max(1, l_{work})$, where

$l_{work} \geq n$ when $norm = 'I'$ or ' 1 ' or ' O '; otherwise, $work$ is not referenced.

Output Parameters

val REAL for slanhb/clanhb

DOUBLE PRECISION for dlanhb/zlanhb

Value returned by the function.

?lansp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix supplied in packed form.

Syntax

```
val = slansp( norm, uplo, n, ap, work )
val = dlansp( norm, uplo, n, ap, work )
val = clansp( norm, uplo, n, ap, work )
val = zlansp( norm, uplo, n, ap, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lansp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix A , supplied in packed form.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine:
= 'M' or 'm':	<i>val</i> = $\max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .
= '1' or 'O' or 'o':	<i>val</i> = $\text{norm}_1(A)$, 1-norm of the matrix A (maximum column sum),
= 'I' or 'i':	<i>val</i> = $\text{norm}_I(A)$, infinity norm of the matrix A (maximum row sum),
= 'F', 'f', 'E' or 'e':	<i>val</i> = $\text{norm}_F(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is supplied. If <i>uplo</i> = 'U': Upper triangular part of A is supplied If <i>uplo</i> = 'L': Lower triangular part of A is supplied.
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, ?lansp is set to zero.
<i>ap</i>	REAL for slansp DOUBLE PRECISION for dlansp COMPLEX for clansp DOUBLE COMPLEX for zlansp Array, DIMENSION ($n(n+1)/2$). The upper or lower triangle of the symmetric matrix A , packed columnwise in a linear array. The j -th column of A is stored in the array <i>ap</i> as follows: if <i>uplo</i> = 'U', $ap(i + (j-1)j/2) = A(i, j)$ for $1 \leq i \leq j$; if <i>uplo</i> = 'L', $ap(i + (j-1)(2n-j)/2) = A(i, j)$ for $j \leq i \leq n$.
<i>work</i>	REAL for slansp and clansp. DOUBLE PRECISION for dlansp and zlansp. Workspace array, DIMENSION ($\max(1, lwork)$), where $lwork \geq n$ when <i>norm</i> = 'I' or '1' or 'O'; otherwise, <i>work</i> is not referenced.

Output Parameters

`val`

REAL for `slansp/clansp`
DOUBLE PRECISION for `dlansp/zlansp`

Value returned by the function.

?lanhp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix supplied in packed form.

Syntax

```
val = clanhp( norm, uplo, n, ap, work )
val = zlanhp( norm, uplo, n, ap, work )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The function `?lanhp` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix A , supplied in packed form.

Input Parameters

`norm`

CHARACTER*1. Specifies the value to be returned by the routine:

- = 'M' or 'm': $val = \max(|A_{ij}|)$, largest absolute value of the matrix A .
- = '1' or 'O' or 'o': $val = \text{norm}_1(A)$, 1-norm of the matrix A (maximum column sum),
- = 'I' or 'i': $val = \text{norm}_I(A)$, infinity norm of the matrix A (maximum row sum),
- = 'F', 'f', 'E' or 'e': $val = \text{norm}_F(A)$, Frobenius norm of the matrix A (square root of sum of squares).

`uplo`

CHARACTER*1.

Specifies whether the upper or lower triangular part of the Hermitian matrix A is supplied.

- If $uplo = 'U'$: Upper triangular part of A is supplied
- If $uplo = 'L'$: Lower triangular part of A is supplied.

`n`

INTEGER. The order of the matrix A .

$n \geq 0$. When $n = 0$, `?lanhp` is set to zero.

`ap`

COMPLEX for `clanhp`.

DOUBLE COMPLEX for `zlanhp`.

Array, DIMENSION ($n(n+1)/2$). The upper or lower triangle of the Hermitian matrix A , packed columnwise in a linear array. The j -th column of A is stored in the array ap as follows:

```
if uplo = 'U', ap(i + (j-1)j/2) = A(i,j) for 1 ≤ i ≤ j;
if uplo = 'L', ap(i + (j-1)(2n-j)/2) = A(i,j) for j ≤ i ≤ n.
```

work

REAL for `clanhp`.

DOUBLE PRECISION for `zlanhp`.

Workspace array, DIMENSION ($\max(1, lwork)$), where

$lwork \geq n$ when $norm = 'I'$ or ' 1 ' or ' O '; otherwise, `work` is not referenced.

Output Parameters

val

REAL for `clanhp`.

DOUBLE PRECISION for `zlanhp`.

Value returned by the function.

?lanst/?lanht

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or complex Hermitian tridiagonal matrix.

Syntax

```
val = slanst( norm, n, d, e )
val = dlanst( norm, n, d, e )
val = clanht( norm, n, d, e )
val = zlanht( norm, n, d, e )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The functions `?lanst/?lanht` return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real symmetric or a complex Hermitian tridiagonal matrix A .

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

```
= 'M' or 'm': val = max(abs(Aij)), largest absolute value of the matrix A.
= '1' or 'O' or 'o': val = norm1(A), 1-norm of the matrix A
(maximum column sum),
```

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),
 = 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

n INTEGER. The order of the matrix *A*.

n ≥ 0. When *n* = 0, ?lanst/?lanht is set to zero.

d REAL for slanst/clanht

DOUBLE PRECISION for dlanst/zlanht

Array, DIMENSION (*n*). The diagonal elements of *A*.

e REAL for slanst

DOUBLE PRECISION for dlanst

COMPLEX for clanht

DOUBLE COMPLEX for zlanht

Array, DIMENSION (*n*-1).

The (*n*-1) sub-diagonal or super-diagonal elements of *A*.

Output Parameters

val REAL for slanst/clanht

DOUBLE PRECISION for dlanst/zlanht

Value returned by the function.

?lansy

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.

Syntax

val = slansy(*norm*, *uplo*, *n*, *a*, *lda*, *work*)

val = dlansy(*norm*, *uplo*, *n*, *a*, *lda*, *work*)

val = clansy(*norm*, *uplo*, *n*, *a*, *lda*, *work*)

val = zlansy(*norm*, *uplo*, *n*, *a*, *lda*, *work*)

C:

```
float LAPACKE_slansy (int matrix_layout, char norm, char uplo, lapack_int n, const
float * a, lapack_int lda);
```

```
double LAPACKE_dlansy (int matrix_layout, char norm, char uplo, lapack_int n, const
double * a, lapack_int lda);
```

```
float LAPACKE_clansy (int matrix_layout, char norm, char uplo, lapack_int n, const
lapack_complex_float * a, lapack_int lda);
```

```
double LAPACKE_zlansy (int matrix_layout, char norm, char uplo, lapack_int n, const
lapack_complex_double * a, lapack_int lda);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lansy returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix A .

Input Parameters

The data types are given for the Fortran interface.

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine:
= 'M' or 'm':	<i>val</i> = $\max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .
= '1' or 'O' or 'o':	<i>val</i> = $\text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),
= 'I' or 'i':	<i>val</i> = $\text{normI}(A)$, infinity norm of the matrix A (maximum row sum),
= 'F', 'f', 'E' or 'e':	<i>val</i> = $\text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is to be referenced. = 'U': Upper triangular part of A is referenced. = 'L': Lower triangular part of A is referenced
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, ?lansy is set to zero.
<i>a</i>	REAL for slansy DOUBLE PRECISION for dlansy COMPLEX for clansy DOUBLE COMPLEX for zlansy Array, DIMENSION (<i>lda,n</i>). The symmetric matrix A . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of <i>a</i> contains the upper triangular part of the matrix A , and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of <i>a</i> contains the lower triangular part of the matrix A , and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . <i>lda</i> $\geq \max(n,1)$.
<i>work</i>	REAL for slansy and clansy.

DOUBLE PRECISION for `dlansy` and `zlansy`.
Workspace array, DIMENSION (`max(1,lwork)`), where
 $lwork \geq n$ when $norm = 'I'$ or $'1'$ or $'O'$; otherwise, `work` is not referenced.

Output Parameters

`val` REAL for `slansy/clansy`
 DOUBLE PRECISION for `dlansy/zlansy`
 Value returned by the function.

?lanhe

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.

Syntax

```
val = clanhe( norm, uplo, n, a, lda, work )
val = zlanhe( norm, uplo, n, a, lda, work )
```

C:

```
float LAPACKE_clanhe (int matrix_layout, char norm, char uplo, lapack_int n, const
lapack_complex_float * a, lapack_int lda);
double LAPACKE_zlanhe (int matrix_layout, char norm, char uplo, lapack_int n, const
lapack_complex_double * a, lapack_int lda);
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The function `?lanhe` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix A .

Input Parameters

The data types are given for the Fortran interface.

<code>norm</code>	CHARACTER*1. Specifies the value to be returned by the routine:
= 'M' or 'm':	<code>val</code> = $\max(A_{ij})$, largest absolute value of the matrix A .
= '1' or 'O' or 'o':	<code>val</code> = $\text{norm}_1(A)$, 1-norm of the matrix A (maximum column sum),
= 'I' or 'i':	<code>val</code> = $\text{norm}_I(A)$, infinity norm of the matrix A (maximum row sum),
= 'F', 'f', 'E' or 'e':	<code>val</code> = $\text{norm}_F(A)$, Frobenius norm of the matrix A (square root of sum of squares).

<i>uplo</i>	CHARACTER*1.
	Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is to be referenced.
	= 'U': Upper triangular part of <i>A</i> is referenced.
	= 'L': Lower triangular part of <i>A</i> is referenced
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$. When $n = 0$, ?lanhe is set to zero.
<i>a</i>	COMPLEX for clanhe. DOUBLE COMPLEX for zlanhe. Array, DIMENSION (<i>lda,n</i>). The Hermitian matrix <i>A</i> . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix <i>A</i> , and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of the matrix <i>A</i> , and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . <i>lda</i> $\geq \max(n, 1)$.
<i>work</i>	REAL for clanhe. DOUBLE PRECISION for zlanhe. Workspace array, DIMENSION ($\max(1, lwork)$), where <i>lwork</i> $\geq n$ when <i>norm</i> = 'I' or '1' or 'O'; otherwise, <i>work</i> is not referenced.

Output Parameters

<i>val</i>	REAL for clanhe. DOUBLE PRECISION for zlanhe. Value returned by the function.
------------	---

?lantb

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular band matrix.

Syntax

```
val = slantb( norm, uplo, diag, n, k, ab, ldab, work )
val = dlantb( norm, uplo, diag, n, k, ab, ldab, work )
val = clantb( norm, uplo, diag, n, k, ab, ldab, work )
val = zlantb( norm, uplo, diag, n, k, ab, ldab, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lantb returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n triangular band matrix A , with ($k + 1$) diagonals.

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine:
= 'M' or 'm':	<i>val</i> = $\max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .
= '1' or 'O' or 'o':	<i>val</i> = $\text{norm}_1(A)$, 1-norm of the matrix A (maximum column sum),
= 'I' or 'i':	<i>val</i> = $\text{norm}_I(A)$, infinity norm of the matrix A (maximum row sum),
= 'F', 'f', 'E' or 'e':	<i>val</i> = $\text{norm}_F(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular.
= 'U':	Upper triangular
= 'L':	Lower triangular.
<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix A is unit triangular.
= 'N':	Non-unit triangular
= 'U':	Unit triangular.
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, ?lantb is set to zero.
<i>k</i>	INTEGER. The number of super-diagonals of the matrix A if <i>uplo</i> = 'U', or the number of sub-diagonals of the matrix A if <i>uplo</i> = 'L'. $k \geq 0$.
<i>ab</i>	REAL for slantb DOUBLE PRECISION for dlantb COMPLEX for clantb DOUBLE COMPLEX for zlantb Array, DIMENSION (<i>ldab</i> , <i>n</i>). The upper or lower triangular band matrix A , stored in the first $k+1$ rows of <i>ab</i> . The j -th column of A is stored in the j -th column of the array <i>ab</i> as follows: if <i>uplo</i> = 'U', <i>ab</i> ($k+1+i-j,j$) = <i>a</i> (<i>i,j</i>) for $\max(1, j-k) \leq i \leq j$; if <i>uplo</i> = 'L', <i>ab</i> ($1+i-j,j$) = <i>a</i> (<i>i,j</i>) for $j \leq i \leq \min(n, j+k)$.

Note that when $\text{diag} = \text{'U'}$, the elements of the array ab corresponding to the diagonal elements of the matrix A are not referenced, but are assumed to be one.

ldab

INTEGER. The leading dimension of the array ab .

$ldab \geq k+1$.

work

REAL for slantb and clantb .

DOUBLE PRECISION for dlantb and zlantb .

Workspace array, DIMENSION ($\max(1, lwork)$), where

$lwork \geq n$ when $\text{norm} = \text{'I'}$; otherwise, work is not referenced.

Output Parameters

val

REAL for $\text{slantb}/\text{clantb}$.

DOUBLE PRECISION for $\text{dlantb}/\text{zlantb}$.

Value returned by the function.

?lantp

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix supplied in packed form.

Syntax

```
val = slantp( norm, uplo, diag, n, ap, work )
val = dlantp( norm, uplo, diag, n, ap, work )
val = clantp( norm, uplo, diag, n, ap, work )
val = zlantp( norm, uplo, diag, n, ap, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lantp returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a triangular matrix A , supplied in packed form.

Input Parameters

norm

CHARACTER*1. Specifies the value to be returned by the routine:

= 'M' or 'm': $val = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A .

= '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),

= 'I' or 'i': *val* = normI(*A*), infinity norm of the matrix *A* (maximum row sum),
 = 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

uplo CHARACTER*1.

Specifies whether the matrix *A* is upper or lower triangular.

= 'U': Upper triangular

= 'L': Lower triangular.

diag CHARACTER*1.

Specifies whether or not the matrix *A* is unit triangular.

= 'N': Non-unit triangular

= 'U': Unit triangular.

n INTEGER. The order of the matrix *A*.

n ≥ 0. When *n* = 0, ?lantp is set to zero.

ap REAL for slantp

DOUBLE PRECISION for dlantp

COMPLEX for clantp

DOUBLE COMPLEX for zlantp

Array, DIMENSION (*n(n+1)/2*).

The upper or lower triangular matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows:

if *uplo* = 'U', *AP*(*i* + (*j*-1)*j/2*) = *a*(*i*, *j*) for $1 \leq i \leq j$;

if *uplo* = 'L', *ap*(*i* + (*j*-1)(2*n*-*j*)/2) = *a*(*i*, *j*) for $j \leq i \leq n$.

Note that when *diag* = 'U', the elements of the array *ap* corresponding to the diagonal elements of the matrix *A* are not referenced, but are assumed to be one.

work REAL for slantp and clantp.

DOUBLE PRECISION for dlantp and zlantp.

Workspace array, DIMENSION (max(1, *lwork*)), where *lwork* ≥ *n* when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val REAL for slantp/clantp.

DOUBLE PRECISION for dlantp/zlantp.

Value returned by the function.

?lantr

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.

Syntax

```
val = slantr( norm, uplo, diag, m, n, a, lda, work )
val = dlantr( norm, uplo, diag, m, n, a, lda, work )
val = clantr( norm, uplo, diag, m, n, a, lda, work )
val = zlantr( norm, uplo, diag, m, n, a, lda, work )
```

C:

```
float LAPACKE_slantr (char * norm, char * uplo, char * diag, lapack_int * m, lapack_int
* n, const float * a, lapack_int * lda, float * work);

double LAPACKE_dlantr (char * norm, char * uplo, char * diag, lapack_int * m,
lapack_int * n, const double * a, lapack_int * lda, double * work);

float LAPACKE_clantr (char * norm, char * uplo, char * diag, lapack_int * m, lapack_int
* n, const lapack_complex_float * a, lapack_int * lda, float * work);

double LAPACKE_zlantr (char * norm, char * uplo, char * diag, lapack_int * m,
lapack_int * n, const lapack_complex_double * a, lapack_int * lda, double * work);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lantr returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix A .

Input Parameters

The data types are given for the Fortran interface.

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(A_{ij})$, largest absolute value of the matrix A . = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower trapezoidal. = 'U': Upper trapezoidal

= 'L': Lower trapezoidal.

Note that A is triangular instead of trapezoidal if $m = n$.

diag

CHARACTER*1.

Specifies whether or not the matrix A has unit diagonal.

= 'N': Non-unit diagonal

= 'U': Unit diagonal.

m

INTEGER. The number of rows of the matrix A . $m \geq 0$, and if *uplo* = 'U', $m \leq n$.

When $m = 0$, *?lantr* is set to zero.

n

INTEGER. The number of columns of the matrix A . $n \geq 0$, and if *uplo* = 'L', $n \leq m$.

When $n = 0$, *?lantr* is set to zero.

a

REAL for *slantr*

DOUBLE PRECISION for *dlantr*

COMPLEX for *clantr*

DOUBLE COMPLEX for *zlantr*

Array, DIMENSION (*lda,n*).

The trapezoidal matrix A (A is triangular if $m = n$).

If *uplo* = 'U', the leading m -by- n upper trapezoidal part of the array *a* contains the upper trapezoidal matrix, and the strictly lower triangular part of A is not referenced.

If *uplo* = 'L', the leading m -by- n lower trapezoidal part of the array *a* contains the lower trapezoidal matrix, and the strictly upper triangular part of A is not referenced. Note that when *diag* = 'U', the diagonal elements of A are not referenced and are assumed to be one.

lda

INTEGER. The leading dimension of the array *a*.

lda $\geq \max(m, 1)$.

work

REAL for *slantr/clantrp*.

DOUBLE PRECISION for *dlantr/zlantr*.

Workspace array, DIMENSION ($\max(1, lwork)$), where

lwork $\geq m$ when *norm* = 'I'; otherwise, *work* is not referenced.

Output Parameters

val

REAL for *slantr/clantrp*.

DOUBLE PRECISION for *dlantr/zlantr*.

Value returned by the function.

?lanv2

Computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form.

Syntax

```

call slanv2( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )
call dlanv2( a, b, c, d, rt1r, rt1i, rt2r, rt2i, cs, sn )

```

Include Files

- Fortran: mkl.fi
 - C: mkl.h

Description

The routine computes the Schur factorization of a real 2-by-2 nonsymmetric matrix in standard form:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} cs & -sn \\ sn & cs \end{bmatrix} \begin{bmatrix} aa & bb \\ cc & dd \end{bmatrix} \begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix}$$

where either

- $cc = 0$ so that aa and dd are real eigenvalues of the matrix, or
 - $aa = dd$ and $bb*cc < 0$, so that $aa \pm \sqrt{bb*cc}$ are complex conjugate eigenvalues.

The routine was adjusted to reduce the risk of cancellation errors, when computing real eigenvalues, and to ensure, if possible, that $\text{abs}(rt1r) \geq \text{abs}(rt2r)$.

Input Parameters

Output Parameters

a, b, c, d	On exit, overwritten by the elements of the standardized Schur form.
$rt1r, rt1i, rt2r, rt2i$	REAL for slanv2 DOUBLE PRECISION for dlanv2. The real and imaginary parts of the eigenvalues. If the eigenvalues are a complex conjugate pair, $rt1i > 0$.
cs, sn	REAL for slanv2 DOUBLE PRECISION for dlanv2. Parameters of the rotation matrix.

?lapll

Measures the linear dependence of two vectors.

Syntax

```
call slapll( n, x, incx, Y, incy, ssmin )
call dlapll( n, x, incx, Y, incy, ssmin )
call clapll( n, x, incx, Y, incy, ssmin )
call zlapll( n, x, incx, Y, incy, ssmin )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

Given two column vectors x and y of length n , let

$A = (x \ y)$ be the n -by-2 matrix.

The routine ?lapll first computes the QR factorization of A as $A = Q^*R$ and then computes the SVD of the 2-by-2 upper triangular matrix R . The smaller singular value of R is returned in $ssmin$, which is used as the measurement of the linear dependency of the vectors x and y .

Input Parameters

n	INTEGER. The length of the vectors x and y .
x	REAL for slapll DOUBLE PRECISION for dlapll COMPLEX for clapll DOUBLE COMPLEX for zlapll Array, DIMENSION ($1+(n-1)incx$). On entry, x contains the n -vector x .
y	REAL for slapll DOUBLE PRECISION for dlapll COMPLEX for clapll DOUBLE COMPLEX for zlapll Array, DIMENSION ($1+(n-1)incy$). On entry, y contains the n -vector y .
$incx$	INTEGER. The increment between successive elements of x ; $incx > 0$.
$incy$	INTEGER. The increment between successive elements of y ; $incy > 0$.

Output Parameters

x	On exit, x is overwritten.
y	On exit, y is overwritten.
$ssmin$	REAL for <code>slapll/clapll</code> DOUBLE PRECISION for <code>dlapll/zlapll</code> The smallest singular value of the n -by-2 matrix $A = (x \ y)$.

?lapmr

Rearranges rows of a matrix as specified by a permutation vector.

Syntax

FORTRAN 77:

```
call slapmr( forwrd, m, n, x, ldx, k )
call dlapmr( forwrd, m, n, x, ldx, k )
call clapmr( forwrd, m, n, x, ldx, k )
call zlapmr( forwrd, m, n, x, ldx, k )
```

Fortran 95:

```
call lapmr( x, k[, forwrd] )
```

C:

```
lapack_int LAPACKE_<?>lapmr (int matrix_layout, lapack_logical forwrd, lapack_int m,  
lapack_int n, <datatype> * x, lapack_int lidx, lapack_int * k);
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The `?lapmr` routine rearranges the rows of the m -by- n matrix X as specified by the permutation $k(1), k(2), \dots, k(m)$ of the integers $1, \dots, m$.

If *forwrd* = .TRUE., forward permutation:

$X(k(i,*))$ is moved to $X(i,*)$ for $i = 1, 2, \dots, m$.

If *forwrd* = .FALSE., backward permutation:

$X(i,*)$ is moved to $X(k(i,*))$ for $i = 1, 2, \dots, m$.

Input Parameters

The data types are given for the Fortran interface.

<i>forwrd</i>	LOGICAL. If <i>forwrd</i> = .TRUE., forward permutation If <i>forwrd</i> = .FALSE., backward permutation
---------------	--

<i>m</i>	INTEGER. The number of rows of the matrix X . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix X . $n \geq 0$.
<i>x</i>	REAL for <code>slapmr</code> DOUBLE PRECISION for <code>dlapmr</code> COMPLEX for <code>clapmr</code> DOUBLE COMPLEX for <code>zlapmr</code> Array, DIMENSION (<i>lidx,n</i>). On entry, the m -by- n matrix X .
<i>lidx</i>	INTEGER. The leading dimension of the array X , $lidx \geq \max(1,m)$.
<i>k</i>	INTEGER. Array, DIMENSION (<i>m</i>). On entry, <i>k</i> contains the permutation vector and is used as internal workspace.

Output Parameters

<i>x</i>	On exit, <i>x</i> contains the permuted matrix X .
<i>k</i>	On exit, <i>k</i> is reset to its original value.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `?lapmr` interface are as follows:

<i>x</i>	Holds the matrix X of size (<i>n, n</i>).
<i>k</i>	Holds the vector of length <i>m</i> .
<i>forwrd</i>	Specifies the permutation. Must be ' <code>.TRUE.</code> ' or ' <code>.FALSE.</code> '.

See Also

[?lapmt](#)

Performs a forward or backward permutation of the columns of a matrix.

?lapmt

Performs a forward or backward permutation of the columns of a matrix.

Syntax

```
call slapmt( forwrd, m, n, x, lidx, k )
call dlapmt( forwrd, m, n, x, lidx, k )
call clapmt( forwrd, m, n, x, lidx, k )
call zlapmt( forwrd, m, n, x, lidx, k )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine ?lapmt rearranges the columns of the m -by- n matrix X as specified by the permutation $k(1), k(2), \dots, k(n)$ of the integers $1, \dots, n$.

If $forwrd = .TRUE..$, forward permutation:

$X(*, k(j))$ is moved to $X(*, j)$ for $j=1, 2, \dots, n$.

If $forwrd = .FALSE..$, backward permutation:

$X(*, j)$ is moved to $X(*, k(j))$ for $j = 1, 2, \dots, n$.

Input Parameters

$forwrd$	LOGICAL.
	If $forwrd = .TRUE..$, forward permutation
	If $forwrd = .FALSE..$, backward permutation
m	INTEGER. The number of rows of the matrix X . $m \geq 0$.
n	INTEGER. The number of columns of the matrix X . $n \geq 0$.
x	REAL for slapmt DOUBLE PRECISION for dlapmt COMPLEX for clapmt DOUBLE COMPLEX for zlapmt Array, DIMENSION ($lIdx, n$). On entry, the m -by- n matrix X .
$lIdx$	INTEGER. The leading dimension of the array X , $lIdx \geq \max(1, m)$.
k	INTEGER. Array, DIMENSION (n). On entry, k contains the permutation vector and is used as internal workspace.

Output Parameters

x	On exit, x contains the permuted matrix X .
k	On exit, k is reset to its original value.

See Also

?lapmr

?lapy2

Returns $\sqrt{x^2 + y^2}$.

Syntax

```
val = slapy2( x, y )
val = dlapy2( x, y )
```

C:

```
<datatype> LAPACKE_<?>lapy2 (<datatype> x, <datatype> y);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lapy2 returns $\sqrt{x^2+y^2}$, avoiding unnecessary overflow or harmful underflow.

Input Parameters

The data types are given for the Fortran interface.

<i>x, y</i>	REAL for slapy2 DOUBLE PRECISION for dlapy2
	Specify the input values <i>x</i> and <i>y</i> .

Output Parameters

<i>val</i>	REAL for slapy2 DOUBLE PRECISION for dlapy2.
	Value returned by the function.

?lapy3

Returns $\sqrt{x^2+y^2+z^2}$.

Syntax

```
val = slapy3( x, y, z )
val = dlapy3( x, y, z )
```

C:

```
<datatype> LAPACKE_<?>lapy3 ( <datatype> x, <datatype> y, <datatype> z );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lapy3 returns $\sqrt{x^2+y^2+z^2}$, avoiding unnecessary overflow or harmful underflow.

Input Parameters

The data types are given for the Fortran interface.

<i>x, y, z</i>	REAL for slapy3 DOUBLE PRECISION for dlapy3
	Specify the input values <i>x, y</i> and <i>z</i> .

Output Parameters

val REAL for slapy3
DOUBLE PRECISION for dlapy3.
Value returned by the function.

?laqgb

Scales a general band matrix, using row and column scaling factors computed by ?gbequ.

Syntax

```

call slaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call dlaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call claqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )
call zlaqgb( m, n, kl, ku, ab, ldab, r, c, rowcnd, colcnd, amax, equed )

```

Include Files

- Fortran: mkl.fi
 - C: mkl.h

Description

The routine equilibrates a general m -by- n band matrix A with k_l subdiagonals and k_u superdiagonals using the row and column scaling factors in the vectors r and c .

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix <i>A</i> . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix <i>A</i> . $n \geq 0$.
<i>k1</i>	INTEGER. The number of subdiagonals within the band of <i>A</i> . $k1 \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of <i>A</i> . $ku \geq 0$.
<i>ab</i>	REAL for <code>slaqgb</code> DOUBLE PRECISION for <code>dlaqgb</code> COMPLEX for <code>claqgb</code> DOUBLE COMPLEX for <code>zlaqgb</code> <p>Array, DIMENSION (<i>ldab,n</i>). On entry, the matrix <i>A</i> in band storage, in rows 1 to $k1+ku+1$. The <i>j</i>-th column of <i>A</i> is stored in the <i>j</i>-th column of the array <i>ab</i> as follows: $ab(ku+1+i-j, j) = A(i, j)$ for $\max(1, j-ku) \leq i \leq \min(m, j+k1)$.</p>
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> . $lda \geq k1+ku+1$.
<i>amax</i>	REAL for <code>slaqgb/claqgb</code>

	DOUBLE PRECISION for dlaqgb/zlaqgb
	Absolute value of largest matrix entry.
<i>r, c</i>	REAL for slaqgb/claqgb
	DOUBLE PRECISION for dlaqgb/zlaqgb
	Arrays <i>r</i> (<i>m</i>), <i>c</i> (<i>n</i>). Contain the row and column scale factors for <i>A</i> , respectively.
<i>rowcnd</i>	REAL for slaqgb/claqgb
	DOUBLE PRECISION for dlaqgb/zlaqgb
	Ratio of the smallest <i>r</i> (<i>i</i>) to the largest <i>r</i> (<i>i</i>).
<i>colcnd</i>	REAL for slaqgb/claqgb
	DOUBLE PRECISION for dlaqgb/zlaqgb
	Ratio of the smallest <i>c</i> (<i>i</i>) to the largest <i>c</i> (<i>i</i>).

Output Parameters

<i>ab</i>	On exit, the equilibrated matrix, in the same storage format as <i>A</i> . See <i>equed</i> for the form of the equilibrated matrix.
<i>equed</i>	CHARACTER*1. Specifies the form of equilibration that was done. If <i>equed</i> = 'N': No equilibration If <i>equed</i> = 'R': Row equilibration, that is, <i>A</i> has been premultiplied by <i>diag(r)</i> . If <i>equed</i> = 'C': Column equilibration, that is, <i>A</i> has been postmultiplied by <i>diag(c)</i> . If <i>equed</i> = 'B': Both row and column equilibration, that is, <i>A</i> has been replaced by <i>diag(r)*A*diag(c)</i> .

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If *rowcnd* < *thresh*, row scaling is done, and if *colcnd* < *thresh*, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, row scaling is done.

?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by ?geequ.

Syntax

```
call slaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call dlaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
call claqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
```

```
call zlaqge( m, n, a, lda, r, c, rowcnd, colcnd, amax, equed )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine equilibrates a general m -by- n matrix A using the row and column scaling factors in the vectors r and c .

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix A . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix A . $n \geq 0$.
<i>a</i>	REAL for slaqge DOUBLE PRECISION for dlagge COMPLEX for claqge DOUBLE COMPLEX for zlagge Array, DIMENSION (<i>lda,n</i>). On entry, the m -by- n matrix A .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(m, 1)$.
<i>r</i>	REAL for slanqge/claqge DOUBLE PRECISION for dlagge/zlagge Array, DIMENSION (<i>m</i>). The row scale factors for A .
<i>c</i>	REAL for slanqge/claqge DOUBLE PRECISION for dlagge/zlagge Array, DIMENSION (<i>n</i>). The column scale factors for A .
<i>rowcnd</i>	REAL for slanqge/claqge DOUBLE PRECISION for dlagge/zlagge Ratio of the smallest $r(i)$ to the largest $r(i)$.
<i>colcnd</i>	REAL for slanqge/claqge DOUBLE PRECISION for dlagge/zlagge Ratio of the smallest $c(i)$ to the largest $c(i)$.
<i>amax</i>	REAL for slanqge/claqge DOUBLE PRECISION for dlagge/zlagge Absolute value of largest matrix entry.

Output Parameters

<i>a</i>	On exit, the equilibrated matrix. See <i>equed</i> for the form of the equilibrated matrix.
<i>equed</i>	CHARACTER*1. Specifies the form of equilibration that was done. If <i>equed</i> = 'N': No equilibration If <i>equed</i> = 'R': Row equilibration, that is, <i>A</i> has been premultiplied by <i>diag(r)</i> . If <i>equed</i> = 'C': Column equilibration, that is, <i>A</i> has been postmultiplied by <i>diag(c)</i> . If <i>equed</i> = 'B': Both row and column equilibration, that is, <i>A</i> has been replaced by <i>diag(r)*A*diag(c)</i> .

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if row or column scaling should be done based on the ratio of the row or column scaling factors. If *rowcnd* < *thresh*, row scaling is done, and if *colcnd* < *thresh*, column scaling is done. *large* and *small* are threshold values used to decide if row scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, row scaling is done.

?laqhb

Scales a Hermetian band matrix, using scaling factors computed by ?pbequ.

Syntax

```
call claqhb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call zlaqhb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine equilibrates a Hermetian band matrix *A* using the scaling factors in the vector *s*.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the band matrix <i>A</i> is stored. If <i>uplo</i> = 'U': upper triangular. If <i>uplo</i> = 'L': lower triangular.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> .

$n \geq 0$.

kd INTEGER. The number of super-diagonals of the matrix A if $uplo = 'U'$, or the number of sub-diagonals if $uplo = 'L'$.

$kd \geq 0$.

ab COMPLEX for `claqhb`

DOUBLE COMPLEX for `zlaghb`

Array, DIMENSION ($ldab, n$). On entry, the upper or lower triangle of the band matrix A , stored in the first $kd+1$ rows of the array. The j -th column of A is stored in the j -th column of the array ab as follows:

if $uplo = 'U'$, $ab(kd+1+i-j, j) = A(i, j)$ for $\max(1, j-kd) \leq i \leq j$;

if $uplo = 'L'$, $ab(1+i-j, j) = A(i, j)$ for $j \leq i \leq \min(n, j+kd)$.

$ldab$ INTEGER. The leading dimension of the array ab .

$ldab \geq kd+1$.

$scond$ REAL for `claqsb`

DOUBLE PRECISION for `zlaqsb`

Ratio of the smallest $s(i)$ to the largest $s(i)$.

$amax$ REAL for `claqsb`

DOUBLE PRECISION for `zlaqsb`

Absolute value of largest matrix entry.

Output Parameters

ab On exit, if $info = 0$, the triangular factor U or L from the Cholesky factorization $A = U^H * U$ or $A = L * L^H$ of the band matrix A , in the same storage format as A .

s REAL for `claqsb`

DOUBLE PRECISION for `zlaqsb`

Array, DIMENSION (n). The scale factors for A .

$equed$ CHARACTER*1.

Specifies whether or not equilibration was done.

If $equed = 'N'$: No equilibration.

If $equed = 'Y'$: Equilibration was done, that is, A has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters $thresh$, $large$, and $small$, which have the following meaning. $thresh$ is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If $scond < thresh$, scaling is done.

The values *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If $\text{amax} > \text{large}$ or $\text{amax} < \text{small}$, scaling is done.

?laqp2

Computes a QR factorization with column pivoting of the matrix block.

Syntax

```
call slaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call dlaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call claqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
call zlaqp2( m, n, offset, a, lda, jpvt, tau, vn1, vn2, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes a QR factorization with column pivoting of the block $A(\text{offset}+1:m, 1:n)$. The block $A(1:\text{offset}, 1:n)$ is accordingly pivoted, but not factorized.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix A . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix A . $n \geq 0$.
<i>offset</i>	INTEGER. The number of rows of the matrix A that must be pivoted but not factorized. $\text{offset} \geq 0$.
<i>a</i>	REAL for slaqp2 DOUBLE PRECISION for dlaqp2 COMPLEX for claqp2 DOUBLE COMPLEX for zlaqp2 Array, DIMENSION (<i>lda,n</i>). On entry, the m -by- n matrix A .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $\text{lda} \geq \max(1, m)$.
<i>jpvt</i>	INTEGER. Array, DIMENSION (<i>n</i>). On entry, if $\text{jpvt}(i) \neq 0$, the <i>i</i> -th column of A is permuted to the front of A^*P (a leading column); if $\text{jpvt}(i) = 0$, the <i>i</i> -th column of A is a free column.
<i>vn1, vn2</i>	REAL for slaqp2/claqp2 DOUBLE PRECISION for dlaqp2/zlaqp2

Arrays, DIMENSION (n) each. Contain the vectors with the partial and exact column norms, respectively.

work REAL for `slaqp2`
 DOUBLE PRECISION for `dlaqp2`
 COMPLEX for `claqp2`
 DOUBLE COMPLEX for `zlaqp2` Workspace array, DIMENSION (n).

Output Parameters

a On exit, the upper triangle of block $A(\text{offset}+1:m, 1:n)$ is the triangular factor obtained; the elements in block $A(\text{offset}+1:m, 1:n)$ below the diagonal, together with the array *tau*, represent the orthogonal matrix Q as a product of elementary reflectors. Block $A(1:\text{offset}, 1:n)$ has been accordingly pivoted, but not factorized.

jpvt On exit, if $\text{jpvt}(i) = k$, then the i -th column of A^*P was the k -th column of A .

tau REAL for `slaqp2`
 DOUBLE PRECISION for `dlaqp2`
 COMPLEX for `claqp2`
 DOUBLE COMPLEX for `zlaqp2`
 Array, DIMENSION ($\min(m, n)$).
 The scalar factors of the elementary reflectors.

vn1, vn2 Contain the vectors with the partial and exact column norms, respectively.

?laqps

Computes a step of QR factorization with column pivoting of a real m -by- n matrix A by using BLAS level 3.

Syntax

```
call slaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call dlaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call claqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
call zlaqps( m, n, offset, nb, kb, a, lda, jpvt, tau, vn1, vn2, auxv, f, ldf )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine computes a step of QR factorization with column pivoting of a real m -by- n matrix A by using BLAS level 3. The routine tries to factorize NB columns from A starting from the row $\text{offset}+1$, and updates all of the matrix with BLAS level 3 routine `?gemm`.

In some cases, due to catastrophic cancellations, ?laqps cannot factorize NB columns. Hence, the actual number of factorized columns is returned in kb .

Block $A(1:offset, 1:n)$ is accordingly pivoted, but not factorized.

Input Parameters

m	INTEGER. The number of rows of the matrix A . $m \geq 0$.
n	INTEGER. The number of columns of the matrix A . $n \geq 0$.
$offset$	INTEGER. The number of rows of A that have been factorized in previous steps.
nb	INTEGER. The number of columns to factorize.
a	REAL for <code>slaqps</code> DOUBLE PRECISION for <code>dlaqps</code> COMPLEX for <code>claqps</code> DOUBLE COMPLEX for <code>zlaqps</code> Array, DIMENSION (lda, n) . On entry, the m -by- n matrix A .
lda	INTEGER. The leading dimension of the array a . $lda \geq \max(1, m)$.
$jpvt$	INTEGER. Array, DIMENSION (n) . If $jpvt(I) = k$ then column k of the full matrix A has been permuted into position i in AP.
$vn1, vn2$	REAL for <code>slaqps/claqps</code> DOUBLE PRECISION for <code>dlaqps/zlaqps</code> Arrays, DIMENSION (n) each. Contain the vectors with the partial and exact column norms, respectively.
$auxv$	REAL for <code>slaqps</code> DOUBLE PRECISION for <code>dlaqps</code> COMPLEX for <code>claqps</code> DOUBLE COMPLEX for <code>zlaqps</code> Array, DIMENSION (nb) . Auxiliary vector.
f	REAL for <code>slaqps</code> DOUBLE PRECISION for <code>dlaqps</code> COMPLEX for <code>claqps</code> DOUBLE COMPLEX for <code>zlaqps</code> Array, DIMENSION (ldf, nb) . For real flavors, matrix $F^T = L^*Y^T*A$. For complex flavors, matrix $F^H = L^*Y^H*A$.

ldf INTEGER. The leading dimension of the array *f*.
ldf $\geq \max(1, n)$.

Output Parameters

kb INTEGER. The number of columns actually factorized.

a On exit, block $A(\text{offset}+1:m, 1:kb)$ is the triangular factor obtained and block $A(1:\text{offset}, 1:n)$ has been accordingly pivoted, but not factorized. The rest of the matrix, block $A(\text{offset}+1:m, kb+1:n)$ has been updated.

jpvt INTEGER array, DIMENSION (*n*). If $jpvt(I) = k$ then column *k* of the full matrix *A* has been permuted into position *i* in AP.

tau REAL for *slaqps*
 DOUBLE PRECISION for *dlaqps*
 COMPLEX for *claqps*
 DOUBLE COMPLEX for *zlaqps*
 Array, DIMENSION (*kb*). The scalar factors of the elementary reflectors.

vn1, *vn2* The vectors with the partial and exact column norms, respectively.

auxv Auxiliary vector.

f Matrix $F' = L^*Y^*A$.

?laqr0

Computes the eigenvalues of a Hessenberg matrix,
 and optionally the matrixes from the Schur
 decomposition.

Syntax

```
call slaqr0( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work,
lwork, info )
call dlaqr0( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work,
lwork, info )
call claqr0( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork,
info )
call zlaqr0( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork,
info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the eigenvalues of a Hessenberg matrix H , and, optionally, the matrices T and Z from the Schur decomposition $H = Z^* T^* Z^H$, where T is an upper quasi-triangular/triangular matrix (the Schur form), and Z is the orthogonal/unitary matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal/unitary matrix Q so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal/unitary matrix Q : $A = Q^* H^* Q^H = (QZ)^* H^* (QZ)^H$.

Input Parameters

<i>wantt</i>	LOGICAL. If <i>wantt</i> = .TRUE., the full Schur form T is required; If <i>wantt</i> = .FALSE., only eigenvalues are required.
<i>wantz</i>	LOGICAL. If <i>wantz</i> = .TRUE., the matrix of Schur vectors Z is required; If <i>wantz</i> = .FALSE., Schur vectors are not required.
<i>n</i>	INTEGER. The order of the Hessenberg matrix H . ($n \geq 0$).
<i>ilo, ihi</i>	INTEGER. It is assumed that H is already upper triangular in rows and columns 1: $ilo-1$ and $ihi+1:n$, and if $ilo > 1$ then $H(ilo, ilo-1) = 0$. ilo and ihi are normally set by a previous call to <code>cgebal</code> , and then passed to <code>cgehrd</code> when the matrix output by <code>cgebal</code> is reduced to Hessenberg form. Otherwise, ilo and ihi should be set to 1 and n , respectively. If $n > 0$, then $1 \leq ilo \leq ihi \leq n$. If $n=0$, then $ilo=1$ and $ihi=0$
<i>h</i>	REAL for <code>slaqr0</code> DOUBLE PRECISION for <code>dlaqr0</code> COMPLEX for <code>claqr0</code> DOUBLE COMPLEX for <code>zlaqr0</code> . Array, DIMENSION (ldh, n), contains the upper Hessenberg matrix H .
<i>ldh</i>	INTEGER. The leading dimension of the array h . $ldh \geq \max(1, n)$.
<i>iloz, ihiz</i>	INTEGER. Specify the rows of Z to which transformations must be applied if <i>wantz</i> is .TRUE., $1 \leq iloz \leq ilo$; $ihi \leq ihiz \leq n$.
<i>z</i>	REAL for <code>slaqr0</code> DOUBLE PRECISION for <code>dlaqr0</code> COMPLEX for <code>claqr0</code> DOUBLE COMPLEX for <code>zlaqr0</code> . Array, DIMENSION (ldz, ihi), contains the matrix Z if <i>wantz</i> is .TRUE.. If <i>wantz</i> is .FALSE., z is not referenced.

<i>ldz</i>	INTEGER. The leading dimension of the array <i>z</i> . If <i>wantz</i> is .TRUE., then <i>ldz</i> $\geq \max(1, ihiz)$. Otherwise, <i>ldz</i> ≥ 1 .
<i>work</i>	REAL for <i>slaqr0</i> DOUBLE PRECISION for <i>dlaqr0</i> COMPLEX for <i>claqr0</i> DOUBLE COMPLEX for <i>zlaqr0</i> . Workspace array with dimension <i>lwork</i> .
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . <i>lwork</i> $\geq \max(1, n)$ is sufficient, but for the optimal performance a greater workspace may be required, typically as large as $6*n$. It is recommended to use the workspace query to determine the optimal workspace size. If <i>lwork</i> =-1, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters <i>n</i> , <i>ilo</i> , and <i>ihiz</i> . The estimate is returned in <i>work</i> (1). No error messages related to the <i>lwork</i> is issued by <i>xerbla</i> . Neither <i>H</i> nor <i>Z</i> are accessed.

Output Parameters

<i>h</i>	If <i>info</i> =0, and <i>wantt</i> is .TRUE., then <i>h</i> contains the upper quasi-triangular/triangular matrix <i>T</i> from the Schur decomposition (the Schur form). If <i>info</i> =0, and <i>wantt</i> is .FALSE., then the contents of <i>h</i> are unspecified on exit. (The output values of <i>h</i> when <i>info</i> > 0 are given under the description of the <i>info</i> parameter below.) The routine may explicitly set <i>h</i> (<i>i</i> , <i>j</i>) for <i>i</i> > <i>j</i> and <i>j</i> =1,2,... <i>ilo</i> -1 or <i>j</i> = <i>ihiz</i> +1, <i>ihiz</i> +2,... <i>n</i> .
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>w</i>	COMPLEX for <i>claqr0</i> DOUBLE COMPLEX for <i>zlaqr0</i> . Arrays, DIMENSION(<i>n</i>). The computed eigenvalues of <i>h</i> (<i>ilo</i> : <i>ihiz</i> , <i>ilo</i> : <i>ihiz</i>) are stored in <i>w</i> (<i>ilo</i> : <i>ihiz</i>). If <i>wantt</i> is .TRUE., then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in <i>h</i> , with <i>w</i> (<i>i</i>) = <i>h</i> (<i>i</i> , <i>i</i>).
<i>wr</i> , <i>wi</i>	REAL for <i>slaqr0</i> DOUBLE PRECISION for <i>dlaqr0</i> Arrays, DIMENSION(<i>ihiz</i>) each. The real and imaginary parts, respectively, of the computed eigenvalues of <i>h</i> (<i>ilo</i> : <i>ihiz</i> , <i>ilo</i> : <i>ihiz</i>) are stored in <i>wr</i> (<i>ilo</i> : <i>ihiz</i>) and <i>wi</i> (<i>ilo</i> : <i>ihiz</i>). If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of <i>wr</i> and <i>wi</i> , say the <i>i</i> -th and (<i>i</i> +1)-th, with <i>wi</i> (<i>i</i>)>0 and <i>wi</i> (<i>i</i> +1)<0. If <i>wantt</i> is .TRUE.,

then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in h , with $wr(i) = h(i,i)$, and if $h(i:i+1, i:i+1)$ is a 2-by-2 diagonal block, then $wi(i)=\sqrt{(-h(i+1,i)*h(i,i+1))}$.

z
If $wantz$ is .TRUE., then $z(il:ih, iloz:ihiz)$ is replaced by $z(il:ih, iloz:ihiz)*U$, where U is the orthogonal/unitary Schur factor of $h(il:ih, il:ih)$.

If $wantz$ is .FALSE., z is not referenced.

(The output values of z when $info > 0$ are given under the description of the $info$ parameter below.)

$info$
INTEGER.
= 0: the execution is successful.
> 0: if $info = i$, then the routine failed to compute all the eigenvalues. Elements $1:ilo-1$ and $i+1:n$ of wr and wi contain those eigenvalues which have been successfully computed.
> 0: if $wantt$ is .FALSE., then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns ilo through $info$ of the final output value of h .
> 0: if $wantt$ is .TRUE., then (initial value of $h)*U = U^*$ (final value of h , where U is an orthogonal/unitary matrix. The final value of h is upper Hessenberg and quasi-triangular/triangular in rows and columns $info+1$ through ih).
> 0: if $wantz$ is .TRUE., then (final value of $z(il:ih, iloz:ihiz))=(initial value of z(il:ih, iloz:ihiz)*U$, where U is the orthogonal/unitary matrix in the previous expression (regardless of the value of $wantt$).
> 0: if $wantz$ is .FALSE., then z is not accessed.

?laqr1

Sets a scalar multiple of the first column of the product of 2-by-2 or 3-by-3 matrix H and specified shifts.

Syntax

```
call slaqr1( n, h, ldh, sr1, sil, sr2, si2, v )
call dlaqr1( n, h, ldh, sr1, sil, sr2, si2, v )
call claqr1( n, h, ldh, s1, s2, v )
call zlaqr1( n, h, ldh, s1, s2, v )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

Given a 2-by-2 or 3-by-3 matrix H , this routine sets v to a scalar multiple of the first column of the product

$K = (H - s1*I)*(H - s2*I)$, or $K = (H - (sr1 + i*si1)*I)*(H - (sr2 + i*si2)*I)$

scaling to avoid overflows and most underflows.

It is assumed that either 1) $sr1 = sr2$ and $si1 = -si2$, or 2) $si1 = si2 = 0$.

This is useful for starting double implicit shift bulges in the QR algorithm.

Input Parameters

n	INTEGER. The order of the matrix H . n must be equal to 2 or 3.
$sr1, si2, sr2, si2$	REAL for <code>slaqr1</code> DOUBLE PRECISION for <code>dlaqr1</code> Shift values that define K in the formula above.
$s1, s2$	COMPLEX for <code>claqr1</code> DOUBLE COMPLEX for <code>zlaqr1</code> . Shift values that define K in the formula above.
h	REAL for <code>slaqr1</code> DOUBLE PRECISION for <code>dlaqr1</code> COMPLEX for <code>claqr1</code> DOUBLE COMPLEX for <code>zlaqr1</code> . Array, DIMENSION (ldh, n), contains 2-by-2 or 3-by-3 matrix H in the formula above.
ldh	INTEGER. The leading dimension of the array h just as declared in the calling routine. $ldh \geq n$.

Output Parameters

v	REAL for <code>slaqr1</code> DOUBLE PRECISION for <code>dlaqr1</code> COMPLEX for <code>claqr1</code> DOUBLE COMPLEX for <code>zlaqr1</code> . Array with dimension (n) . A scalar multiple of the first column of the matrix K in the formula above.
-----	--

?laqr2

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```

call slaqqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr,
si, v, ldv, nh, t, ldt, nv, ldwv, work, lwork )

call dlaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr,
si, v, ldv, nh, t, ldt, nv, ldwv, work, lwork )

call claqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh,
v, ldv, nh, t, ldt, nv, ldwv, work, lwork )

call zlaqr2( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh,
v, ldv, nh, t, ldt, nv, ldwv, work, lwork )

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine accepts as input an upper Hessenberg matrix H and performs an orthogonal/unitary similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output H has been overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal/unitary similarity transformation of H . It is to be hoped that the final version of H has many zero subdiagonal entries.

This subroutine is identical to ?laqr3 except that it avoids recursion by calling ?lahqr instead of ?laqr4.

Input Parameters

wantt

LOGICAL.

If *wantt* = .TRUE., then the Hessenberg matrix H is fully updated so that the quasi-triangular/triangular Schur factor may be computed (in cooperation with the calling subroutine).

If *wantt* = .FALSE., then only enough of H is updated to preserve the eigenvalues.

wantz

LOGICAL.

If *wantz* = .TRUE., then the orthogonal/unitary matrix Z is updated so that the orthogonal/unitary Schur factor may be computed (in cooperation with the calling subroutine).

If *wantz* = .FALSE., then Z is not referenced.

n

INTEGER. The order of the Hessenberg matrix H and (if *wantz* = .TRUE.) the order of the orthogonal/unitary matrix Z .

ktop

INTEGER.

It is assumed that either *ktop*=1 or $h(ktop,ktop-1)=0$. *ktop* and *kbot* together determine an isolated block along the diagonal of the Hessenberg matrix.

kbot

INTEGER.

It is assumed without a check that either $kbot=n$ or $h(kbot+1, kbot)=0$. $ktop$ and $kbot$ together determine an isolated block along the diagonal of the Hessenberg matrix.

nw

INTEGER.

Size of the deflation window. $1 \leq nw \leq (kbot-ktop+1)$.*h*REAL **for** slaqr2DOUBLE PRECISION **for** dlaqr2COMPLEX **for** claqr2DOUBLE COMPLEX **for** zlaqr2.Array, DIMENSION ($l dh, n$), on input the initial n -by- n section of *h* stores the Hessenberg matrix *H* undergoing aggressive early deflation.*ldh*INTEGER. The leading dimension of the array *h* just as declared in the calling subroutine. $ldh \geq n$.*iloz, ihiz*INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE.. $1 \leq iloz \leq ihiz \leq n$.*z*REAL **for** slaqr2DOUBLE PRECISION **for** dlaqr2COMPLEX **for** claqr2DOUBLE COMPLEX **for** zlaqr2.Array, DIMENSION ($l dz, n$), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., then *z* is not referenced.*ldz*INTEGER. The leading dimension of the array *z* just as declared in the calling subroutine. $ldz \geq 1$.*v*REAL **for** slaqr2DOUBLE PRECISION **for** dlaqr2COMPLEX **for** claqr2DOUBLE COMPLEX **for** zlaqr2.Workspace array with dimension ($l dv, nw$). An nw -by- nw work array.*ldv*INTEGER. The leading dimension of the array *v* just as declared in the calling subroutine. $ldv \geq nw$.*nh*INTEGER. The number of column of *t*. $nh \geq nw$.*t*REAL **for** slaqr2DOUBLE PRECISION **for** dlaqr2COMPLEX **for** claqr2DOUBLE COMPLEX **for** zlaqr2.Workspace array with dimension ($l dt, nw$).

<i>ldt</i>	INTEGER. The leading dimension of the array <i>t</i> just as declared in the calling subroutine. $ldt \geq nw$.
<i>nv</i>	INTEGER. The number of rows of work array <i>wv</i> available for workspace. $nv \geq nw$.
<i>wv</i>	REAL for <i>slaqr2</i> DOUBLE PRECISION for <i>dlaqr2</i> COMPLEX for <i>claqr2</i> DOUBLE COMPLEX for <i>zlaqr2</i> . Workspace array with dimension $(ldwv, nw)$.
<i>ldwv</i>	INTEGER. The leading dimension of the array <i>wv</i> just as declared in the calling subroutine. $ldwv \geq nw$.
<i>work</i>	REAL for <i>slaqr2</i> DOUBLE PRECISION for <i>dlaqr2</i> COMPLEX for <i>claqr2</i> DOUBLE COMPLEX for <i>zlaqr2</i> . Workspace array with dimension <i>lwork</i> .
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . $lwork=2 * nw$ is sufficient, but for the optimal performance a greater workspace may be required. If <i>lwork=-1</i> , then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters <i>n</i> , <i>nw</i> , <i>ktop</i> , and <i>kbot</i> . The estimate is returned in <i>work(1)</i> . No error messages related to the <i>lwork</i> is issued by <i>xerbla</i> . Neither <i>H</i> nor <i>Z</i> are accessed.

Output Parameters

<i>h</i>	On output <i>h</i> has been transformed by an orthogonal/unitary similarity transformation, perturbed, and returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.
<i>work(1)</i>	On exit <i>work(1)</i> is set to an estimate of the optimal value of <i>lwork</i> for the given values of the input parameters <i>n</i> , <i>nw</i> , <i>ktop</i> , and <i>kbot</i> .
<i>z</i>	If <i>wantz</i> is .TRUE., then the orthogonal/unitary similarity transformation is accumulated into <i>z(iloz:ihiz, ilo:ichi)</i> from the right. If <i>wantz</i> is .FALSE., then <i>z</i> is unreferenced.
<i>nd</i>	INTEGER. The number of converged eigenvalues uncovered by the routine.
<i>ns</i>	INTEGER. The number of unconverged, that is approximate eigenvalues returned in <i>sr</i> , <i>si</i> or in <i>sh</i> that may be used as shifts by the calling subroutine.
<i>sh</i>	COMPLEX for <i>claqr2</i>

DOUBLE COMPLEX for zlaqr2.

Arrays, DIMENSION (kbot).

The approximate eigenvalues that may be used for shifts are stored in the $sh(kbot-nd-ns+1)$ through the $sh(kbot-nd)$.

The converged eigenvalues are stored in the $sh(kbot-nd+1)$ through the $sh(kbot)$.

sr, si

REAL for slaqr2

DOUBLE PRECISION for dlaqr2

Arrays, DIMENSION (kbot) each.

The real and imaginary parts of the approximate eigenvalues that may be used for shifts are stored in the $sr(kbot-nd-ns+1)$ through the $sr(kbot-nd)$, and $si(kbot-nd-ns+1)$ through the $si(kbot-nd)$, respectively.

The real and imaginary parts of converged eigenvalues are stored in the $sr(kbot-nd+1)$ through the $sr(kbot)$, and $si(kbot-nd+1)$ through the $si(kbot)$, respectively.

?laqr3

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

```
call slaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr,
si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
call dlaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sr,
si, v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
call claqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh,
v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
call zlaqr3( wantt, wantz, n, ktop, kbot, nw, h, ldh, iloz, ihiz, z, ldz, ns, nd, sh,
v, ldv, nh, t, ldt, nv, wv, ldwv, work, lwork )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine accepts as input an upper Hessenberg matrix H and performs an orthogonal/unitary similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output H has been overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal/unitary similarity transformation of H . It is to be hoped that the final version of H has many zero subdiagonal entries.

Input Parameters

<i>wantt</i>	LOGICAL. If <i>wantt</i> = .TRUE., then the Hessenberg matrix <i>H</i> is fully updated so that the quasi-triangular/triangular Schur factor may be computed (in cooperation with the calling subroutine). If <i>wantt</i> = .FALSE., then only enough of <i>H</i> is updated to preserve the eigenvalues.
<i>wantz</i>	LOGICAL. If <i>wantz</i> = .TRUE., then the orthogonal/unitary matrix <i>Z</i> is updated so that the orthogonal/unitary Schur factor may be computed (in cooperation with the calling subroutine). If <i>wantz</i> = .FALSE., then <i>Z</i> is not referenced.
<i>n</i>	INTEGER. The order of the Hessenberg matrix <i>H</i> and (if <i>wantz</i> = .TRUE.) the order of the orthogonal/unitary matrix <i>Z</i> .
<i>ktop</i>	INTEGER. It is assumed that either <i>ktop</i> =1 or <i>h</i> (<i>ktop</i> , <i>ktop</i> -1)=0. <i>ktop</i> and <i>kbot</i> together determine an isolated block along the diagonal of the Hessenberg matrix.
<i>kbot</i>	INTEGER. It is assumed without a check that either <i>kbot</i> = <i>n</i> or <i>h</i> (<i>kbot</i> +1, <i>kbot</i>)=0. <i>ktop</i> and <i>kbot</i> together determine an isolated block along the diagonal of the Hessenberg matrix.
<i>nw</i>	INTEGER. Size of the deflation window. $1 \leq nw \leq (kbot - ktop + 1)$.
<i>h</i>	REAL for <i>slaqr3</i> DOUBLE PRECISION for <i>dlaqr3</i> COMPLEX for <i>claqr3</i> DOUBLE COMPLEX for <i>zlaqr3</i> . Array, DIMENSION (<i>ldh</i> , <i>n</i>), on input the initial <i>n</i> -by- <i>n</i> section of <i>h</i> stores the Hessenberg matrix <i>H</i> undergoing aggressive early deflation.
<i>ldh</i>	INTEGER. The leading dimension of the array <i>h</i> just as declared in the calling subroutine. $ldh \geq n$.
<i>iloz</i> , <i>ihiz</i>	INTEGER. Specify the rows of <i>Z</i> to which transformations must be applied if <i>wantz</i> is .TRUE.. $1 \leq iloz \leq ihiz \leq n$.
<i>z</i>	REAL for <i>slaqr3</i> DOUBLE PRECISION for <i>dlaqr3</i> COMPLEX for <i>claqr3</i> DOUBLE COMPLEX for <i>zlaqr3</i> .

Array, **DIMENSION** (*ldz*, *n*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., then *z* is not referenced.

ldz INTEGER. The leading dimension of the array *z* just as declared in the calling subroutine. *ldz* $\geq 1.$

v REAL **for** *slaqr3*

DOUBLE PRECISION **for** *dlaqr3*

COMPLEX **for** *claqr3*

DOUBLE COMPLEX **for** *zlaqr3*.

Workspace array with dimension (*ldv*, *nw*). An *nw*-by-*nw* work array.

ldv INTEGER. The leading dimension of the array *v* just as declared in the calling subroutine. *ldv* $\geq nw.$

nh INTEGER. The number of column of *t*. *nh* $\geq nw.$

t REAL **for** *slaqr3*

DOUBLE PRECISION **for** *dlaqr3*

COMPLEX **for** *claqr3*

DOUBLE COMPLEX **for** *zlaqr3*.

Workspace array with dimension (*ldt*, *nw*).

ldt INTEGER. The leading dimension of the array *t* just as declared in the calling subroutine. *ldt* $\geq nw.$

nv INTEGER. The number of rows of work array *wv* available for workspace. *nv* $\geq nw.$

wv REAL **for** *slaqr3*

DOUBLE PRECISION **for** *dlaqr3*

COMPLEX **for** *claqr3*

DOUBLE COMPLEX **for** *zlaqr3*.

Workspace array with dimension (*ldwv*, *nw*).

ldwv INTEGER. The leading dimension of the array *wv* just as declared in the calling subroutine. *ldwv* $\geq nw.$

work REAL **for** *slaqr3*

DOUBLE PRECISION **for** *dlaqr3*

COMPLEX **for** *claqr3*

DOUBLE COMPLEX **for** *zlaqr3*.

Workspace array with dimension *lwork*.

lwork INTEGER. The dimension of the array *work*.

`lwork=2*nw`) is sufficient, but for the optimal performance a greater workspace may be required.

If `lwork=-1`, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters `n`, `nw`, `ktop`, and `kbot`. The estimate is returned in `work(1)`. No error messages related to the `lwork` is issued by `xerbla`. Neither `H` nor `Z` are accessed.

Output Parameters

`h`

On output `h` has been transformed by an orthogonal/unitary similarity transformation, perturbed, and returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

`work(1)`

On exit `work(1)` is set to an estimate of the optimal value of `lwork` for the given values of the input parameters `n`, `nw`, `ktop`, and `kbot`.

`z`

If `wantz` is .TRUE., then the orthogonal/unitary similarity transformation is accumulated into `z(iloz:ihiz, ilo:ichi)` from the right.

If `wantz` is .FALSE., then `z` is unreferenced.

`nd`

INTEGER. The number of converged eigenvalues uncovered by the routine.

`ns`

INTEGER. The number of unconverged, that is approximate eigenvalues returned in `sr`, `si` or in `sh` that may be used as shifts by the calling subroutine.

`sh`

COMPLEX for `claqr3`

DOUBLE COMPLEX for `zlaqr3`.

Arrays, DIMENSION (`kbot`).

The approximate eigenvalues that may be used for shifts are stored in the `sh(kbot-nd-ns+1)` through the `sh(kbot-nd)`.

The converged eigenvalues are stored in the `sh(kbot-nd+1)` through the `sh(kbot)`.

`sr, si`

REAL for `slaqr3`

DOUBLE PRECISION for `dlaqr3`

Arrays, DIMENSION (`kbot`) each.

The real and imaginary parts of the approximate eigenvalues that may be used for shifts are stored in the `sr(kbot-nd-ns+1)` through the `sr(kbot-nd)`, and `si(kbot-nd-ns+1)` through the `si(kbot-nd)`, respectively.

The real and imaginary parts of converged eigenvalues are stored in the `sr(kbot-nd+1)` through the `sr(kbot)`, and `si(kbot-nd+1)` through the `si(kbot)`, respectively.

?laqr4

Computes the eigenvalues of a Hessenberg matrix, and optionally the matrices from the Schur decomposition.

Syntax

```

call slaqqr4( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work,
lwork, info )

call dlaqr4( wantt, wantz, n, ilo, ihi, h, ldh, wr, wi, iloz, ihiz, z, ldz, work,
lwork, info )

call claqr4( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork,
info )

call zlaqr4( wantt, wantz, n, ilo, ihi, h, ldh, w, iloz, ihiz, z, ldz, work, lwork,
info )

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the eigenvalues of a Hessenberg matrix H , and, optionally, the matrices T and Z from the Schur decomposition $H = Z^* T^* Z^H$, where T is an upper quasi-triangular/triangular matrix (the Schur form), and Z is the orthogonal/unitary matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal/unitary matrix Q so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal/unitary matrix Q : $A = Q^* H^* Q^H = (QZ)^* H^* (QZ)^H$.

This routine implements one level of recursion for ?laqr0. It is a complete implementation of the small bulge multi-shift QR algorithm. It may be called by ?laqr0 and, for large enough deflation window size, it may be called by ?laqr3. This routine is identical to ?laqr0 except that it calls ?laqr2 instead of ?laqr3.

Input Parameters

<code>wantt</code>	LOGICAL. If <code>wantt</code> = .TRUE., the full Schur form T is required; If <code>wantt</code> = .FALSE., only eigenvalues are required.
<code>wantz</code>	LOGICAL. If <code>wantz</code> = .TRUE., the matrix of Schur vectors Z is required; If <code>wantz</code> = .FALSE., Schur vectors are not required.
<code>n</code>	INTEGER. The order of the Hessenberg matrix H . ($n \geq 0$).
<code>ilo, ihi</code>	INTEGER. It is assumed that H is already upper triangular in rows and columns $1:ilo-1$ and $ihi+1:n$, and if $ilo > 1$ then $h(ilo, ilo-1) = 0$. <code>ilo</code> and <code>ihi</code> are normally set by a previous call to cgebal, and then passed to cgehrd when the matrix output by cgebal is reduced to Hessenberg form. Otherwise, <code>ilo</code> and <code>ihi</code> should be set to 1 and n , respectively. If $n > 0$, then $1 \leq ilo \leq ihi \leq n$. If $n=0$, then <code>ilo=1</code> and <code>ihi=0</code>

h REAL for `slaqr4`
 DOUBLE PRECISION for `dlaqr4`
 COMPLEX for `claqr4`
 DOUBLE COMPLEX for `zlaqr4`.
 Array, DIMENSION (*ldh, n*), contains the upper Hessenberg matrix *H*.

ldh INTEGER. The leading dimension of the array *h*. *ldh* $\geq \max(1, n)$.

ilo, ihiz INTEGER. Specify the rows of *Z* to which transformations must be applied if *wantz* is .TRUE., $1 \leq ilo \leq ihi; ihi \leq ihiz \leq n$.

z REAL for `slaqr4`
 DOUBLE PRECISION for `dlaqr4`
 COMPLEX for `claqr4`
 DOUBLE COMPLEX for `zlaqr4`.
 Array, DIMENSION (*ldz, ihi*), contains the matrix *Z* if *wantz* is .TRUE.. If *wantz* is .FALSE., *z* is not referenced.

ldz INTEGER. The leading dimension of the array *z*.
 If *wantz* is .TRUE., then *ldz* $\geq \max(1, ihiz)$. Otherwise, *ldz* ≥ 1 .

work REAL for `slaqr4`
 DOUBLE PRECISION for `dlaqr4`
 COMPLEX for `claqr4`
 DOUBLE COMPLEX for `zlaqr4`.
 Workspace array with dimension *lwork*.

lwork INTEGER. The dimension of the array *work*.
 lwork $\geq \max(1, n)$ is sufficient, but for the optimal performance a greater workspace may be required, typically as large as $6*n$.
 It is recommended to use the workspace query to determine the optimal workspace size. If *lwork=-1*, then the routine performs a workspace query: it estimates the optimal workspace size for the given values of the input parameters *n, ilo*, and *ihi*. The estimate is returned in *work(1)*. No error messages related to the *lwork* is issued by `xerbla`. Neither *H* nor *Z* are accessed.

Output Parameters

h If *info=0*, and *wantt* is .TRUE., then *h* contains the upper quasi-triangular/triangular matrix *T* from the Schur decomposition (the Schur form).
 If *info=0*, and *wantt* is .FALSE., then the contents of *h* are unspecified on exit.
 (The output values of *h* when *info > 0* are given under the description of the *info* parameter below.)

The routines may explicitly set $h(i,j)$ for $i > j$ and $j=1, 2, \dots, ilo-1$ or $j=ih+1, ih+2, \dots, n$.

work(1)

On exit *work(1)* contains the minimum value of *lwork* required for optimum performance.

w

COMPLEX for *claqr4*

DOUBLE COMPLEX for *zlaqr4*.

Arrays, DIMENSION(*n*). The computed eigenvalues of $h(il:ih, il:ih)$ are stored in *w(il:ih)*. If *wantt* is .TRUE., then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with $w(i) = h(i,i)$.

wr, wi

REAL for *slaqr4*

DOUBLE PRECISION for *dlaqr4*

Arrays, DIMENSION(*ih*) each. The real and imaginary parts, respectively, of the computed eigenvalues of $h(il:ih, il:ih)$ are stored in the *wr(il:ih)* and *wi(il:ih)*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and (*i*+1)-th, with *wi(i)* > 0 and *wi(i+1)* < 0. If *wantt* is .TRUE., then the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*, with *wr(i) = h(i,i)*, and if $h(i:i+1, i:i+1)$ is a 2-by-2 diagonal block, then *wi(i)=sqrt(-h(i+1,i)*h(i,i+1))*.

z

If *wantz* is .TRUE., then $z(il:ih, iloz:ihiz)$ is replaced by $z(il:ih, iloz:ihiz)*U$, where *U* is the orthogonal/unitary Schur factor of $h(il:ih, il:ih)$.

If *wantz* is .FALSE., *z* is not referenced.

(The output values of *z* when *info* > 0 are given under the description of the *info* parameter below.)

info

INTEGER.

= 0: the execution is successful.

> 0: if *info* = *i*, then the routine failed to compute all the eigenvalues.

Elements 1:*ilo*-1 and *i*+1:*n* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

> 0: if *wantt* is .FALSE., then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns *ilo* through *info* of the final output value of *h*.

> 0: if *wantt* is .TRUE., then (initial value of *h*)**U* = *U** (final value of *h*), where *U* is an orthogonal/unitary matrix. The final value of *h* is upper Hessenberg and quasi-triangular/triangular in rows and columns *info*+1 through *ih*.

> 0: if *wantz* is .TRUE., then (final value of *z(il:ih, iloz:ihiz)*)=(initial value of *z(il:ih, iloz:ihiz)*)**U*, where *U* is the orthogonal/unitary matrix in the previous expression (regardless of the value of *wantt*).

> 0: if *wantz* is .FALSE., then *z* is not accessed.

?laqr5

Performs a single small-bulge multi-shift QR sweep.

Syntax

```
call slaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh, iloz, ihiz,
z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
call dlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh, iloz, ihiz,
z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
call claqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, s, h, ldh, iloz, ihiz, z,
ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
call zlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, s, h, ldh, iloz, ihiz, z,
ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This auxiliary routine called by ?laqr0 performs a single small-bulge multi-shift QR sweep.

Input Parameters

<i>wantt</i>	LOGICAL.
	<i>wantt</i> = .TRUE. if the quasi-triangular/triangular Schur factor is computed.
	<i>wantt</i> is set to .FALSE. otherwise.
<i>wantz</i>	LOGICAL.
	<i>wantz</i> = .TRUE. if the orthogonal/unitary Schur factor is computed.
	<i>wantz</i> is set to .FALSE. otherwise.
<i>kacc22</i>	INTEGER. Possible values are 0, 1, or 2.
	Specifies the computation mode of far-from-diagonal orthogonal updates.
	= 0: the routine does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries.
	= 1: the routine accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.
	= 2: the routine accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.
<i>n</i>	INTEGER. The order of the Hessenberg matrix <i>H</i> upon which the routine operates.

<i>ktop, kbot</i>	INTEGER. It is assumed without a check that either <i>ktop</i> =1 or <i>h</i> (<i>ktop</i> , <i>ktop</i> -1)=0, and either <i>kbot</i> = <i>n</i> or <i>h</i> (<i>kbot</i> +1, <i>kbot</i>)=0.
<i>nshfts</i>	INTEGER. Number of simultaneous shifts, must be positive and even.
<i>sr, si</i>	REAL for slaqr5 DOUBLE PRECISION for dlaqr5 Arrays, DIMENSION (<i>nshfts</i>) each. <i>sr</i> contains the real parts and <i>si</i> contains the imaginary parts of the <i>nshfts</i> shifts of origin that define the multi-shift QR sweep.
<i>s</i>	COMPLEX for claqr5 DOUBLE COMPLEX for zlaqr5. Arrays, DIMENSION (<i>nshfts</i>). <i>s</i> contains the shifts of origin that define the multi-shift QR sweep.
<i>h</i>	REAL for slaqr5 DOUBLE PRECISION for dlaqr5 COMPLEX for claqr5 DOUBLE COMPLEX for zlaqr5. Array, DIMENSION (<i>ldh</i> , <i>n</i>), on input contains the Hessenberg matrix.
<i>ldh</i>	INTEGER. The leading dimension of the array <i>h</i> just as declared in the calling routine. <i>ldh</i> $\geq \max(1, n)$.
<i>iloz, ihiz</i>	INTEGER. Specify the rows of <i>Z</i> to which transformations must be applied if <i>wantz</i> is .TRUE.. $1 \leq iloz \leq ihiz \leq n$.
<i>z</i>	REAL for slaqr5 DOUBLE PRECISION for dlaqr5 COMPLEX for claqr5 DOUBLE COMPLEX for zlaqr5. Array, DIMENSION (<i>ldz</i> , <i>ihi</i>), contains the matrix <i>Z</i> if <i>wantz</i> is .TRUE.. If <i>wantz</i> is .FALSE., then <i>z</i> is not referenced.
<i>ldz</i>	INTEGER. The leading dimension of the array <i>z</i> just as declared in the calling routine. <i>ldz</i> $\geq n$.
<i>v</i>	REAL for slaqr5 DOUBLE PRECISION for dlaqr5 COMPLEX for claqr5 DOUBLE COMPLEX for zlaqr5. Workspace array with dimension (<i>ldv</i> , <i>nshfts</i> /2).

<i>ldv</i>	INTEGER. The leading dimension of the array <i>v</i> just as declared in the calling routine. $ldv \geq 3$.
<i>u</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> DOUBLE COMPLEX for <code>zlaqr5</code> . Workspace array with dimension $(ldu, 3*nshfts-3)$.
<i>ldu</i>	INTEGER. The leading dimension of the array <i>u</i> just as declared in the calling routine. $ldu \geq 3*nshfts-3$.
<i>nh</i>	INTEGER. The number of column in the array <i>wh</i> available for workspace. $nh \geq 1$.
<i>wh</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> DOUBLE COMPLEX for <code>zlaqr5</code> . Workspace array with dimension $(ldwh, nh)$
<i>ldwh</i>	INTEGER. The leading dimension of the array <i>wh</i> just as declared in the calling routine. $ldwh \geq 3*nshfts-3$
<i>nv</i>	INTEGER. The number of rows of the array <i>wv</i> available for workspace. $nv \geq 1$.
<i>wv</i>	REAL for <code>slaqr5</code> DOUBLE PRECISION for <code>dlaqr5</code> COMPLEX for <code>claqr5</code> DOUBLE COMPLEX for <code>zlaqr5</code> . Workspace array with dimension $(ldwv, 3*nshfts-3)$.
<i>ldwv</i>	INTEGER. The leading dimension of the array <i>wv</i> just as declared in the calling routine. $ldwv \geq nv$.

Output Parameters

<i>sr, si</i>	On output, may be reordered.
<i>h</i>	On output a multi-shift QR Sweep with shifts $sr(j)+i*si(j)$ or $s(j)$ is applied to the isolated diagonal block in rows and columns <i>ktop</i> through <i>kbot</i> .
<i>z</i>	If <i>wantz</i> is .TRUE., then the QR Sweep orthogonal/unitary similarity transformation is accumulated into $z(iloz:ihiz, ilo:ihj)$ from the right. If <i>wantz</i> is .FALSE., then <i>z</i> is unreferenced.

?laqsb

Scales a symmetric band matrix, using scaling factors computed by ?pbequ.

Syntax

```
call slaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call dlaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call claqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
call zlaqsb( uplo, n, kd, ab, ldab, s, scond, amax, equed )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine equilibrates a symmetric band matrix A using the scaling factors in the vector s .

Input Parameters

uplo CHARACTER*1.
 Specifies whether the upper or lower triangular part of the symmetric matrix A is stored.
 If *uplo* = 'U': upper triangular.
 If *uplo* = 'L': lower triangular.

n INTEGER. The order of the matrix A .
 $n \geq 0$.

kd INTEGER. The number of super-diagonals of the matrix A if *uplo* = 'U', or the number of sub-diagonals if *uplo* = 'L'.
 $kd \geq 0$.

ab REAL for slaqsb
 DOUBLE PRECISION for dlaqsb
 COMPLEX for claqsb
 DOUBLE COMPLEX for zlaqsb
 Array, DIMENSION (*ldab,n*). On entry, the upper or lower triangle of the symmetric band matrix A , stored in the first $kd+1$ rows of the array. The j -th column of A is stored in the j -th column of the array *ab* as follows:
 if *uplo* = 'U', $ab(kd+1+i-j,j) = A(i,j)$ for $\max(1, j-kd) \leq i \leq j$;
 if *uplo* = 'L', $ab(1+i-j,j) = A(i,j)$ for $j \leq i \leq \min(n, j+kd)$.

ldab INTEGER. The leading dimension of the array *ab*.
 $ldab \geq kd+1$.

<i>s</i>	REAL for <code>slaqsb/claqsb</code> DOUBLE PRECISION for <code>dlaqsb/zlaqsb</code> Array, DIMENSION (<i>n</i>). The scale factors for <i>A</i> .
<i>scond</i>	REAL for <code>slaqsb/claqsb</code> DOUBLE PRECISION for <code>dlaqsb/zlaqsb</code> Ratio of the smallest <i>s</i> (<i>i</i>) to the largest <i>s</i> (<i>i</i>).
<i>amax</i>	REAL for <code>slaqsb/claqsb</code> DOUBLE PRECISION for <code>dlaqsb/zlaqsb</code> Absolute value of largest matrix entry.

Output Parameters

<i>ab</i>	On exit, if <i>info</i> = 0, the triangular factor <i>U</i> or <i>L</i> from the Cholesky factorization of the band matrix <i>A</i> that can be <i>A</i> = <i>U</i> ^T * <i>U</i> or <i>A</i> = <i>L</i> * <i>L</i> ^T for real flavors and <i>A</i> = <i>U</i> ^H * <i>U</i> or <i>A</i> = <i>L</i> * <i>L</i> ^H for complex flavors, in the same storage format as <i>A</i> .
<i>equed</i>	CHARACTER*1. Specifies whether or not equilibration was done. If <i>equed</i> = 'N': No equilibration. If <i>equed</i> = 'Y': Equilibration was done, that is, <i>A</i> has been replaced by <i>diag</i> (<i>s</i>)* <i>A</i> * <i>diag</i> (<i>s</i>).

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*, which have the following meaning. *thresh* is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If *scond* < *thresh*, scaling is done. *large* and *small* are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If *amax* > *large* or *amax* < *small*, scaling is done.

?laqsp

Scales a symmetric/Hermitian matrix in packed storage, using scaling factors computed by ?ppequ.

Syntax

```
call slaqsp( uplo, n, ap, s, scond, amax, equed )
call dlaqsp( uplo, n, ap, s, scond, amax, equed )
call claqsp( uplo, n, ap, s, scond, amax, equed )
call zlaqsp( uplo, n, ap, s, scond, amax, equed )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine ?laqsp equilibrates a symmetric matrix A using the scaling factors in the vector s .

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is stored. If $uplo = 'U'$: upper triangular. If $uplo = 'L'$: lower triangular.
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>ap</i>	REAL for <code>slaqsp</code> DOUBLE PRECISION for <code>dlaqsp</code> COMPLEX for <code>claqsp</code> DOUBLE COMPLEX for <code>zlaqsp</code> Array, DIMENSION $(n(n+1)/2)$. On entry, the upper or lower triangle of the symmetric matrix A , packed columnwise in a linear array. The j -th column of A is stored in the array <i>ap</i> as follows: <code>if uplo = 'U', ap(i + (j-1)j/2) = A(i,j) for 1 ≤ i ≤ j;</code> <code>if uplo = 'L', ap(i + (j-1)(2n-j)/2) = A(i,j) for j≤i≤n.</code>
<i>s</i>	REAL for <code>slaqsp/claqsp</code> DOUBLE PRECISION for <code>dlaqsp/zlaqsp</code> Array, DIMENSION (<i>n</i>). The scale factors for A .
<i>scond</i>	REAL for <code>slaqsp/claqsp</code> DOUBLE PRECISION for <code>dlaqsp/zlaqsp</code> Ratio of the smallest $s(i)$ to the largest $s(i)$.
<i>amax</i>	REAL for <code>slaqsp/claqsp</code> DOUBLE PRECISION for <code>dlaqsp/zlaqsp</code> Absolute value of largest matrix entry.

Output Parameters

<i>ap</i>	On exit, the equilibrated matrix: $\text{diag}(s) * A * \text{diag}(s)$, in the same storage format as A .
<i>equed</i>	CHARACTER*1. Specifies whether or not equilibration was done. If $equed = 'N'$: No equilibration.

If $\text{equed} = \text{'Y'}$: Equilibration was done, that is, A has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters thresh , large , and small , which have the following meaning. thresh is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If $scond < \text{thresh}$, scaling is done. large and small are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If $\text{amax} > \text{large}$ or $\text{amax} < \text{small}$, scaling is done.

?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by ?poequ.

Syntax

```
call slaqsy( uplo, n, a, lda, s, scond, amax, equed )
call dlaqsy( uplo, n, a, lda, s, scond, amax, equed )
call claqsy( uplo, n, a, lda, s, scond, amax, equed )
call zlaqsy( uplo, n, a, lda, s, scond, amax, equed )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine equilibrates a symmetric matrix A using the scaling factors in the vector s .

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is stored. If $\text{uplo} = \text{'U'}$: upper triangular. If $\text{uplo} = \text{'L'}$: lower triangular.
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>a</i>	REAL for slaqsy DOUBLE PRECISION for dlaqsy COMPLEX for claqsy DOUBLE COMPLEX for zlaqsy Array, DIMENSION (<i>lda,n</i>). On entry, the symmetric matrix A . If $\text{uplo} = \text{'U'}$, the leading n -by- n upper triangular part of <i>a</i> contains the upper triangular part of the matrix A , and the strictly lower triangular part of <i>a</i> is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

lda	INTEGER. The leading dimension of the array a .
	$lda \geq \max(n, 1)$.
s	REAL for <code>slaqsy/claqsy</code> DOUBLE PRECISION for <code>dlaqsy/zlaqsy</code> Array, DIMENSION (n). The scale factors for A .
$scond$	REAL for <code>slaqsy/claqsy</code> DOUBLE PRECISION for <code>dlaqsy/zlaqsy</code> Ratio of the smallest $s(i)$ to the largest $s(i)$.
$amax$	REAL for <code>slaqsy/claqsy</code> DOUBLE PRECISION for <code>dlaqsy/zlaqsy</code> Absolute value of largest matrix entry.

Output Parameters

a	On exit, if $equed = 'Y'$, the equilibrated matrix: $\text{diag}(s) * A * \text{diag}(s)$.
$equed$	CHARACTER*1. Specifies whether or not equilibration was done. If $equed = 'N'$: No equilibration. If $equed = 'Y'$: Equilibration was done, i.e., A has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters $thresh$, $large$, and $small$, which have the following meaning. $thresh$ is a threshold value used to decide if scaling should be based on the ratio of the scaling factors. If $scond < thresh$, scaling is done. $large$ and $small$ are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If $amax > large$ or $amax < small$, scaling is done.

?laqtr

Solves a real quasi-triangular system of equations, or a complex quasi-triangular system of special form, in real arithmetic.

Syntax

```
call slaqtr( ltran, lreal, n, t, ldt, b, w, scale, x, work, info )
call dlaqtr( ltran, lreal, n, t, ldt, b, w, scale, x, work, info )
```

Include Files

- Fortran: `mkl.fi`

- C: mkl.h

Description

The routine ?laqtr solves the real quasi-triangular system

`op(T) * p = scale*c, if lreal = .TRUE.`

or the complex quasi-triangular systems

`op(T + iB) * (p+iq) = scale*(c+id), if lreal = .FALSE.`

in real arithmetic, where T is upper quasi-triangular.

If `lreal = .FALSE.`, then the first diagonal block of T must be 1-by-1, B is the specially structured matrix

$\text{op}(A) = A \text{ or } A^T$, A^T denotes the transpose of matrix A .

On input,

www.ijerph.org | ISSN: 1660-4601 | DOI: 10.3390/ijerph17030894

This routine is designed for the condition number estimation in routine ?trsna.

Input Parameters

ltran

LOGICAL.

On entry, *Itran* specifies the option of conjugate transpose:

$= .\text{FALSE}., \text{op}(T + iB) = T + iB,$

$$= .\text{TRUE}. , \text{op}(T + iB) = (T + iB)^T.$$

1real

LOGICAL.

On entry, *lreal* specifies the input matrix structure:

`≡ .FALSE., the input is complex`

= TRUE the input is real

n

INTEGER

On entry, n specifies the order of $T + iB$, $n \geq 0$.

七

BEAT for slagter

DOUBLE PRECISION **for** `dlaqtr`

Array, dimension (ldt, n). On entry, t contains a matrix in Schur canonical form. If $lreal = .FALSE.$, then the first diagonal block of t must be 1-by-1.

ldt

INTEGER. The leading dimension of the matrix T .

$ldt \geq \max(1, n)$.

b

REAL **for** `slaqtr`

DOUBLE PRECISION **for** `dlaqtr`

Array, dimension (n). On entry, b contains the elements to form the matrix B as described above. If $lreal = .TRUE.$, b is not referenced.

w

REAL **for** `slaqtr`

DOUBLE PRECISION **for** `dlaqtr`

On entry, w is the diagonal element of the matrix B .

If $lreal = .TRUE.$, w is not referenced.

x

REAL **for** `slaqtr`

DOUBLE PRECISION **for** `dlaqtr`

Array, dimension ($2n$). On entry, x contains the right hand side of the system.

$work$

REAL **for** `slaqtr`

DOUBLE PRECISION **for** `dlaqtr`

Workspace array, dimension (n).

Output Parameters

$scale$

REAL **for** `slaqtr`

DOUBLE PRECISION **for** `dlaqtr`

On exit, $scale$ is the scale factor.

x

On exit, X is overwritten by the solution.

$info$

INTEGER.

If $info = 0$: successful exit.

If $info = 1$: the some diagonal 1-by-1 block has been perturbed by a small number $smin$ to keep nonsingularity.

If $info = 2$: the some diagonal 2-by-2 block has been perturbed by a small number in `?laln2` to keep nonsingularity.

NOTE

For higher speed, this routine does not check the inputs for errors.

?lar1v

Computes the (scaled) r -th column of the inverse of the submatrix in rows $b1$ through bn of tridiagonal matrix.

Syntax

```
call slar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcorr, work )
call dlar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcorr, work )
call clar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcorr, work )
call zlar1v( n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcorr, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lar1v computes the (scaled) r -th column of the inverse of the submatrix in rows $b1$ through bn of the tridiagonal matrix $L^*D^*L^T - \lambda^*I$. When λ is close to an eigenvalue, the computed vector is an accurate eigenvector. Usually, r corresponds to the index where the eigenvector is largest in magnitude.

The following steps accomplish this computation :

- Stationary qd transform, $L^*D^*L^T - \lambda^*I = L(+)^*D(+)^*L(+)^T$
- Progressive qd transform, $L^*D^*L^T - \lambda^*I = U(-)^*D(-)^*U(-)^T$,
- Computation of the diagonal elements of the inverse of $L^*D^*L^T - \lambda^*I$ by combining the above transforms, and choosing r as the index where the diagonal of the inverse is (one of the) largest in magnitude.
- Computation of the (scaled) r -th column of the inverse using the twisted factorization obtained by combining the top part of the stationary and the bottom part of the progressive transform.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix $L^*D^*L^T$.
<i>b1</i>	INTEGER. First index of the submatrix of $L^*D^*L^T$.
<i>bn</i>	INTEGER. Last index of the submatrix of $L^*D^*L^T$.
<i>lambda</i>	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v
	The shift. To compute an accurate eigenvector, <i>lambda</i> should be a good approximation to an eigenvalue of $L^*D^*L^T$.
<i>l</i>	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v

Array, DIMENSION (n-1).

The (n-1) subdiagonal elements of the unit bidiagonal matrix L , in elements 1 to n-1.

d

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

Array, DIMENSION (n).

The n diagonal elements of the diagonal matrix D .

ld

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

Array, DIMENSION (n-1).

The $n-1$ elements $L_i^*D_i$.

lld

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

Array, DIMENSION (n-1).

The $n-1$ elements $L_i^*L_i^*D_i$.

pivmin

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

The minimum pivot in the Sturm sequence.

gaptol

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

Tolerance that indicates when eigenvector entries are negligible with respect to their contribution to the residual.

z

REAL for slar1v

DOUBLE PRECISION for dlar1v

COMPLEX for clar1v

DOUBLE COMPLEX for zlar1v

Array, DIMENSION (n). All entries of *z* must be set to 0.

wantnc

LOGICAL.

Specifies whether *negcnt* has to be computed.

r

INTEGER.

The twist index for the twisted factorization used to compute *z*. On input, $0 \leq r \leq n$. If *r* is input as 0, *r* is set to the index where $(L^*D^*L^T - \lambda^*I)^{-1}$ is largest in magnitude. If $1 \leq r \leq n$, *r* is unchanged.

work

REAL for slar1v/clar1v

DOUBLE PRECISION for dlar1v/zlar1v

Workspace array, DIMENSION (4*n).

Output Parameters

<i>z</i>	REAL for slar1v DOUBLE PRECISION for dlar1v COMPLEX for clar1v DOUBLE COMPLEX for zlar1v Array, DIMENSION (n). The (scaled) <i>r</i> -th column of the inverse. <i>z(r)</i> is returned to be 1.
<i>negcnt</i>	INTEGER. If <i>wantnc</i> is .TRUE. then <i>negcnt</i> = the number of pivots < <i>pivmin</i> in the matrix factorization $L^*D^*L^T$, and <i>negcnt</i> = -1 otherwise.
<i>ztz</i>	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v The square of the 2-norm of <i>z</i> .
<i>mingma</i>	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v The reciprocal of the largest (in magnitude) diagonal element of the inverse of $L^*D^*L^T - \lambda I$.
<i>r</i>	On output, <i>r</i> is the twist index used to compute <i>z</i> . Ideally, <i>r</i> designates the position of the maximum entry in the eigenvector.
<i>isuppz</i>	INTEGER. Array, DIMENSION (2). The support of the vector in <i>Z</i> , that is, the vector <i>z</i> is nonzero only in elements <i>isuppz</i> (1) through <i>isuppz</i> (2).
<i>nrminv</i>	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v Equals $1/\sqrt{ztz}$.
<i>resid</i>	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v The residual of the FP vector. <i>resid</i> = ABS(<i>mingma</i>)/sqrt(<i>ztz</i>).
<i>rqcorr</i>	REAL for slar1v/clar1v DOUBLE PRECISION for dlar1v/zlar1v The Rayleigh Quotient correction to <i>lambda</i> . <i>rqcorr</i> = <i>mingma</i> / <i>ztz</i> .

?lar2v

Applies a vector of plane rotations with real cosines and real/complex sines from both sides to a sequence of 2-by-2 symmetric/Hermitian matrices.

Syntax

```
call slar2v( n, x, y, z, incx, c, s, incc )
call dlar2v( n, x, y, z, incx, c, s, incc )
call clar2v( n, x, y, z, incx, c, s, incc )
call zlar2v( n, x, y, z, incx, c, s, incc )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lar2v applies a vector of real/complex plane rotations with real cosines from both sides to a sequence of 2-by-2 real symmetric or complex Hermitian matrices, defined by the elements of the vectors x, y and z. For $i = 1, 2, \dots, n$

$$\begin{bmatrix} x_i & z_i \\ \text{conj}(z_i) & y_i \end{bmatrix} := \begin{bmatrix} c(i) & \text{conj}(s(i)) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i & z_i \\ \text{conj}(z_i) & y_i \end{bmatrix} \begin{bmatrix} c(i) & -\text{conj}(s(i)) \\ s(i) & c(i) \end{bmatrix}$$

Input Parameters

<i>n</i>	INTEGER. The number of plane rotations to be applied.
<i>x, y, z</i>	REAL for slar2v DOUBLE PRECISION for dlar2v COMPLEX for clar2v DOUBLE COMPLEX for zlar2v Arrays, DIMENSION (1+(n-1)*incx) each. Contain the vectors x, y and z, respectively. For all flavors of ?lar2v, elements of x and y are assumed to be real.
<i>incx</i>	INTEGER. The increment between elements of x, y, and z. $incx > 0$.
<i>c</i>	REAL for slar2v/clar2v DOUBLE PRECISION for dlar2v/zlar2v Array, DIMENSION (1+(n-1)*incc). The cosines of the plane rotations.
<i>s</i>	REAL for slar2v DOUBLE PRECISION for dlar2v COMPLEX for clar2v

DOUBLE COMPLEX for zlar2v
Array, DIMENSION (1+(n-1)*incc). The sines of the plane rotations.
incc INTEGER. The increment between elements of *c* and *s*. *incc* > 0.

Output Parameters

<i>x, y, z</i>	Vectors <i>x, y</i> and <i>z</i> , containing the results of transform.
----------------	---

?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

```
call slarf( side, m, n, v, incv, tau, c, ldc, work )
call dlarf( side, m, n, v, incv, tau, c, ldc, work )
call clarf( side, m, n, v, incv, tau, c, ldc, work )
call zlarf( side, m, n, v, incv, tau, c, ldc, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine applies a real/complex elementary reflector H to a real/complex m -by- n matrix C , from either the left or the right. H is represented in one of the following forms:

- $H = I - \tau v^* v^T$
where τ is a real scalar and v is a real vector.
If $\tau = 0$, then H is taken to be the unit matrix.
- $H = I - \tau v^* v^H$
where τ is a complex scalar and v is a complex vector.
If $\tau = 0$, then H is taken to be the unit matrix. For clarf/zlarf, to apply H^H (the conjugate transpose of H), supply conjg(τ) instead of τ .

Input Parameters

<i>side</i>	CHARACTER*1. If <i>side</i> = 'L': form $H*C$ If <i>side</i> = 'R': form $C*H$.
<i>m</i>	INTEGER. The number of rows of the matrix C .
<i>n</i>	INTEGER. The number of columns of the matrix C .
<i>v</i>	REAL for slarf

DOUBLE PRECISION **for** dlarf
 COMPLEX **for** clarf
 DOUBLE COMPLEX **for** zlarf
Array, DIMENSION
 $(1 + (m-1) * \text{abs}(incv))$ if $\text{side} = 'L'$ or
 $(1 + (n-1) * \text{abs}(incv))$ if $\text{side} = 'R'$. The vector v in the representation of H . v is not used if $\tau = 0$.

incv INTEGER. The increment between elements of v .
 $incv \neq 0$.

tau REAL **for** slarf
 DOUBLE PRECISION **for** dlarf
 COMPLEX **for** clarf
 DOUBLE COMPLEX **for** zlarf
 The value τ in the representation of H .

c REAL **for** slarf
 DOUBLE PRECISION **for** dlarf
 COMPLEX **for** clarf
 DOUBLE COMPLEX **for** zlarf
Array, DIMENSION (ldc, n).
 On entry, the m -by- n matrix C .

ldc INTEGER. The leading dimension of the array c .
 $ldc \geq \max(1, m)$.

work REAL **for** slarf
 DOUBLE PRECISION **for** dlarf
 COMPLEX **for** clarf
 DOUBLE COMPLEX **for** zlarf
Workspace array, DIMENSION
 (n) if $\text{side} = 'L'$ or
 (m) if $\text{side} = 'R'$.

Output Parameters

c On exit, C is overwritten by the matrix $H*C$ if $\text{side} = 'L'$, or $C*H$ if $\text{side} = 'R'$.

?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

```
call slarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc, work,
ldwork )

call dlarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc, work,
ldwork )

call clarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc, work,
ldwork )

call zlarfb( side, trans, direct, storev, m, n, k, v, ldv, t, ldt, c, ldc, work,
ldwork )
```

C:

```
lapack_int LAPACKE_<?>larfb (int matrix_layout, char side, char trans, char direct,
char storev, lapack_int m, lapack_int n, lapack_int k, const <datatype> * v,
lapack_int ldv, const <datatype> * t, lapack_int ldt, <datatype> * c, lapack_int ldc);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The real flavors of the routine ?larfb apply a real block reflector H or its transpose H^T to a real m -by- n matrix C from either left or right.

The complex flavors of the routine ?larfb apply a complex block reflector H or its conjugate transpose H^H to a complex m -by- n matrix C from either left or right.

Input Parameters

The data types are given for the Fortran interface.

<i>side</i>	CHARACTER*1. If <i>side</i> = 'L': apply H or H^T for real flavors and H or H^H for complex flavors from the left. If <i>side</i> = 'R': apply H or H^T for real flavors and H or H^H for complex flavors from the right.
<i>trans</i>	CHARACTER*1. If <i>trans</i> = 'N': apply H (No transpose). If <i>trans</i> = 'C': apply H^H (Conjugate transpose). If <i>trans</i> = 'T': apply H^T (Transpose).
<i>direct</i>	CHARACTER*1. Indicates how H is formed from a product of elementary reflectors If <i>direct</i> = 'F': $H = H(1)*H(2)*\dots*H(k)$ (forward)

If `direct = 'B'`: $H = H(k) * \dots * H(2) * H(1)$ (backward)

`storev`

CHARACTER*1.

Indicates how the vectors which define the elementary reflectors are stored:

If `storev = 'C'`: Column-wise

If `storev = 'R'`: Row-wise

`m`

INTEGER. The number of rows of the matrix C .

`n`

INTEGER. The number of columns of the matrix C .

`k`

INTEGER. The order of the matrix T (equal to the number of elementary reflectors whose product defines the block reflector).

`v`

REAL **for** `slarfb`

DOUBLE PRECISION **for** `dlarfb`

COMPLEX **for** `clarfb`

DOUBLE COMPLEX **for** `zlarfb`

Array, DIMENSION

(ldv, k) if `storev = 'C'`

(ldv, m) if `storev = 'R'` and `side = 'L'`

(ldv, n) if `storev = 'R'` and `side = 'R'`

The matrix v . See *Application Notes* below.

`ldv`

INTEGER. The leading dimension of the array v .

If `storev = 'C'` and `side = 'L'`, $ldv \geq \max(1, m)$;

if `storev = 'C'` and `side = 'R'`, $ldv \geq \max(1, n)$;

if `storev = 'R'`, $ldv \geq k$.

`t`

REAL **for** `slarfb`

DOUBLE PRECISION **for** `dlarfb`

COMPLEX **for** `clarfb`

DOUBLE COMPLEX **for** `zlarfb`

Array, DIMENSION (ldt, k).

Contains the triangular k -by- k matrix T in the representation of the block reflector.

`ldt`

INTEGER. The leading dimension of the array t .

$ldt \geq k$.

`c`

REAL **for** `slarfb`

DOUBLE PRECISION **for** `dlarfb`

COMPLEX **for** `clarfb`

DOUBLE COMPLEX for zlarfb
Array, DIMENSION (*ldc,n*).
 On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*.
 $ldc \geq \max(1, m)$.

work REAL for slarfb
 DOUBLE PRECISION for dlarfb
 COMPLEX for clarfb
 DOUBLE COMPLEX for zlarfb
Workspace array, DIMENSION (*ldwork, k*).
ldwork INTEGER. The leading dimension of the array *work*.
If *side* = 'L', *ldwork* $\geq \max(1, n)$;
if *side* = 'R', *ldwork* $\geq \max(1, m)$.

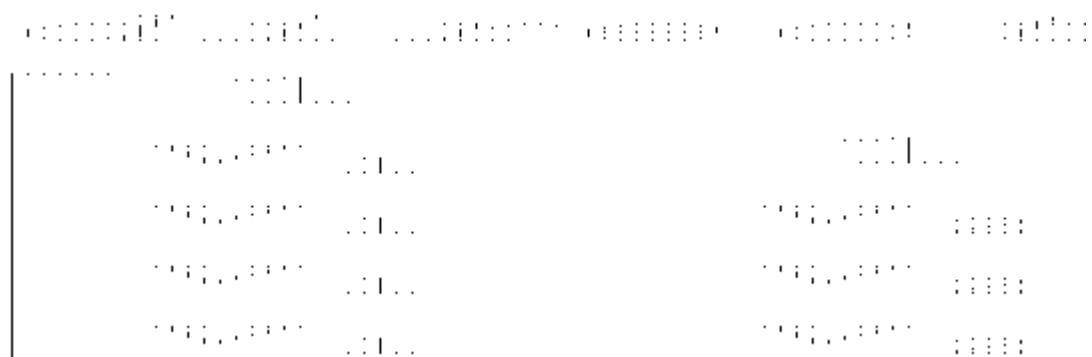
Output Parameters

c On exit, *c* is overwritten by the product of the following:

- H^*C , or H^T*C , or C^*H , or C^*H^T for real flavors
- H^*C , or $H^{H*}C$, or C^*H , or C^*H^H for complex flavors

Application Notes

The shape of the matrix *V* and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with *n* = 5 and *k* = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.



direct = 'B' and storev = 'C': direct = 'B' and storev = 'R':

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ 1 & v_3 \\ & & 1 \end{bmatrix}$$

$$\begin{bmatrix} v_1 & v_1 & 1 \\ v_2 & v_2 & v_2 & 1 \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

?larfg

Generates an elementary reflector (Householder matrix).

Syntax

```
call slarfg( n, alpha, x, incx, tau )
call dlarfg( n, alpha, x, incx, tau )
call clarfg( n, alpha, x, incx, tau )
call zlarfg( n, alpha, x, incx, tau )
```

C:

```
lapack_int LAPACKE_<?>larfg (lapack_int n, <datatype> * alpha, <datatype> * x,
lapack_int incx, <datatype> * tau);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?larfg generates a real/complex elementary reflector H of order n , such that

$$H^* \begin{bmatrix} \alpha \\ x \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, H^T * H = I, \quad \text{for real flavors and}$$

$$H^X * \begin{bmatrix} \alpha \\ x \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, H^X * H = I, \quad \text{for complex flavors,}$$

where α and β are scalars (with β real for all flavors), and x is an $(n-1)$ -element real/complex vector. H is represented in the form

$$H = I - \tau \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v^T \end{bmatrix} \quad \text{for real flavors and}$$

$$H = I - \tau \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v' \end{bmatrix} \quad \text{for complex flavors,}$$

where τ is a real/complex scalar and v is a real/complex $(n-1)$ -element vector, respectively. Note that for `clarfg/zlarfg`, H is not Hermitian.

If the elements of x are all zero (and, for complex flavors, α is real), then $\tau = 0$ and H is taken to be the unit matrix.

Otherwise, $1 \leq \tau \leq 2$ (for real flavors), or

$1 \leq \text{Re}(\tau) \leq 2$ and $\text{abs}(\tau - 1) \leq 1$ (for complex flavors).

Input Parameters

The data types are given for the Fortran interface.

<i>n</i>	INTEGER. The order of the elementary reflector.
<i>alpha</i>	REAL for <code>slarfg</code> DOUBLE PRECISION for <code>dlarfg</code> COMPLEX for <code>clarfg</code> DOUBLE COMPLEX for <code>zlarfg</code> On entry, the value α .
<i>x</i>	REAL for <code>slarfg</code> DOUBLE PRECISION for <code>dlarfg</code> COMPLEX for <code>clarfg</code> DOUBLE COMPLEX for <code>zlarfg</code> Array, DIMENSION $(1+(n-2)*\text{abs}(incx))$. On entry, the vector x .
<i>incx</i>	INTEGER. The increment between elements of x . $incx > 0$.

Output Parameters

<i>alpha</i>	On exit, it is overwritten with the value β .
<i>x</i>	On exit, it is overwritten with the vector v .
<i>tau</i>	REAL for <code>slarfg</code> DOUBLE PRECISION for <code>dlarfg</code> COMPLEX for <code>clarfg</code> DOUBLE COMPLEX for <code>zlarfg</code> The value τ .

?larfgp

Generates an elementary reflector (Householder matrix) with non-negative beta .

Syntax

```
call slarfgp( n, alpha, x, incx, tau )
call dlarfgp( n, alpha, x, incx, tau )
call clarfgp( n, alpha, x, incx, tau )
```

```
call zlarfgp( n, alpha, x, incx, tau )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?larfgp generates a real/complex elementary reflector H of order n , such that

$$H^* \begin{bmatrix} \alpha \\ x \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, H^T * H = I, \quad \text{for real flavors and}$$

$$H^X * \begin{bmatrix} \alpha \\ x \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, H^X * H = I, \quad \text{for complex flavors,}$$

where α and β are scalars (with β real and non-negative for all flavors), and x is an $(n-1)$ -element real/complex vector. H is represented in the form

$$H = I - \tau * \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v^T \end{bmatrix} \quad \text{for real flavors and}$$

$$H = I - \tau * \begin{bmatrix} 1 \\ v \end{bmatrix} * \begin{bmatrix} 1 & v' \end{bmatrix} \quad \text{for complex flavors,}$$

where τ is a real/complex scalar and v is a real/complex $(n-1)$ -element vector. Note that for c/zlarfgp, H is not Hermitian.

If the elements of x are all zero (and, for complex flavors, α is real), then $\tau = 0$ and H is taken to be the unit matrix.

Otherwise, $1 \leq \tau \leq 2$ (for real flavors), or

$1 \leq \operatorname{Re}(\tau) \leq 2$ and $\operatorname{abs}(\tau-1) \leq 1$ (for complex flavors).

Input Parameters

n INTEGER. The order of the elementary reflector.

alpha REAL for slarfgp

DOUBLE PRECISION for dlarfgp

COMPLEX for clarfgp

DOUBLE COMPLEX for zlarfgp

On entry, the value alpha.

x REAL for s

DOUBLE PRECISION for dlarfgp

COMPLEX for clarfgp

DOUBLE COMPLEX for zlarfgp

Array, DIMENSION $(1+(n-2)*\operatorname{abs}(incx))$.

On entry, the vector x .

incx INTEGER.

The increment between elements of x . $incx > 0$.

Output Parameters

alpha On exit, it is overwritten with the value *beta*.

x On exit, it is overwritten with the vector v .

tau REAL for slarfpg
DOUBLE PRECISION for dlarfpg
COMPLEX for clarfpg
DOUBLE COMPLEX for zlarfpg
The value *tau*.

?larft

Forms the triangular factor T of a block reflector $H = I - V^*T^*V^{**}H$.

Syntax

```
call slarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call dlarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call clarft( direct, storev, n, k, v, ldv, tau, t, ldt )
call zlarft( direct, storev, n, k, v, ldv, tau, t, ldt )
```

C:

```
lapack_int LAPACKE_<?>larft (int matrix_layout, char direct, char storev, lapack_int n,
lapack_int k, const <datatype> * v, lapack_int ldv, const <datatype> * tau, <datatype>
* t, lapack_int ldt);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?larft forms the triangular factor T of a real/complex block reflector H of order n , which is defined as a product of k elementary reflectors.

If *direct* = 'F', $H = H(1)*H(2)* \dots *H(k)$ and T is upper triangular;

If *direct* = 'B', $H = H(k)* \dots *H(2)*H(1)$ and T is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector $H(i)$ is stored in the i -th column of the array v , and $H = I - V^*T^*V^T$ (for real flavors) or $H = I - V^*T^*V^H$ (for complex flavors).

If *storev* = 'R', the vector which defines the elementary reflector $H(i)$ is stored in the i -th row of the array v , and $H = I - V^T*T^*V$ (for real flavors) or $H = I - V^H*T^*V$ (for complex flavors).

Input Parameters

The data types are given for the Fortran interface.

<i>direct</i>	CHARACTER*1. Specifies the order in which the elementary reflectors are multiplied to form the block reflector: = 'F': $H = H(1) * H(2) * \dots * H(k)$ (forward) = 'B': $H = H(k) * \dots * H(2) * H(1)$ (backward)
<i>storev</i>	CHARACTER*1. Specifies how the vectors which define the elementary reflectors are stored (see also <i>Application Notes</i> below): = 'C': column-wise = 'R': row-wise.
<i>n</i>	INTEGER. The order of the block reflector H . $n \geq 0$.
<i>k</i>	INTEGER. The order of the triangular factor T (equal to the number of elementary reflectors). $k \geq 1$.
<i>v</i>	REAL for slarft DOUBLE PRECISION for dlarft COMPLEX for clarft DOUBLE COMPLEX for zlarft Array, DIMENSION (<i>ldv</i> , <i>k</i>) if <i>storev</i> = 'C' or (<i>ldv</i> , <i>n</i>) if <i>storev</i> = 'R'. The matrix V .
<i>ldv</i>	INTEGER. The leading dimension of the array <i>v</i> . If <i>storev</i> = 'C', <i>ldv</i> $\geq \max(1, n)$; if <i>storev</i> = 'R', <i>ldv</i> $\geq k$.
<i>tau</i>	REAL for slarft DOUBLE PRECISION for dlarft COMPLEX for clarft DOUBLE COMPLEX for zlarft Array, DIMENSION (<i>k</i>). <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$.
<i>ldt</i>	INTEGER. The leading dimension of the output array <i>t</i> . <i>ldt</i> $\geq k$.

Output Parameters

<i>t</i>	REAL for slarft
----------	-----------------

DOUBLE PRECISION for dlarft
 COMPLEX for clarft
 DOUBLE COMPLEX for zlarft
 Array, DIMENSION (*ldt*,*k*). The *k*-by-*k* triangular factor *T* of the block reflector. If *direct* = 'F', *T* is upper triangular; if *direct* = 'B', *T* is lower triangular. The rest of the array is not used.

v The matrix *V*.

Application Notes

The shape of the matrix *V* and the storage of the vectors which define the *H*(*i*) is best illustrated by the following example with *n* = 5 and *k* = 3. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'B' and *storev* = 'C': *direct* = 'B' and *storev* = 'R':

$$\begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ 1 & v_3 \\ & 1 \end{bmatrix} \quad \begin{bmatrix} v_1 & v_1 & 1 \\ v_2 & v_2 & v_2 & 1 \\ v_3 & v_3 & v_3 & v_3 & 1 \end{bmatrix}$$

?larfx

Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order less than or equal to 10.

Syntax

```
call slarfx( side, m, n, v, tau, c, ldc, work )
call dlarfx( side, m, n, v, tau, c, ldc, work )
call clarfx( side, m, n, v, tau, c, ldc, work )
```

```
call zlarfx( side, m, n, v, tau, c, ldc, work )
```

C:

```
lapack_int LAPACKE_<?>larfx (int matrix_layout, char side, lapack_int m, lapack_int n,
const <datatype> * v, <datatype> tau, <datatype> * c, lapack_int ldc, <datatype> *
work);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?larfx applies a real/complex elementary reflector H to a real/complex m -by- n matrix C , from either the left or the right.

H is represented in the following forms:

- $H = I - \tau u^* v^T$, where τ is a real scalar and v is a real vector.
- $H = I - \tau u^* v^* v^H$, where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then H is taken to be the unit matrix.

Input Parameters

The data types are given for the Fortran interface.

<i>side</i>	CHARACTER*1. If <i>side</i> = 'L': form H^*C If <i>side</i> = 'R': form C^*H .
<i>m</i>	INTEGER. The number of rows of the matrix C .
<i>n</i>	INTEGER. The number of columns of the matrix C .
<i>v</i>	REAL for slarfx DOUBLE PRECISION for dlarfx COMPLEX for clarfx DOUBLE COMPLEX for zlarfx Array, DIMENSION (<i>m</i>) if <i>side</i> = 'L' or (<i>n</i>) if <i>side</i> = 'R'. The vector <i>v</i> in the representation of H .
<i>tau</i>	REAL for slarfx DOUBLE PRECISION for dlarfx COMPLEX for clarfx DOUBLE COMPLEX for zlarfx The value <i>tau</i> in the representation of H .
<i>c</i>	REAL for slarfx

DOUBLE PRECISION **for** dlarfx
 COMPLEX **for** clarfx
 DOUBLE COMPLEX **for** zlarfx

Array, DIMENSION (*ldc,n*). On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*. *lda* $\geq (1,m).$

work REAL **for** slarfx
 DOUBLE PRECISION **for** dlarfx
 COMPLEX **for** clarfx
 DOUBLE COMPLEX **for** zlarfx

Workspace array, DIMENSION
(n) if *side* = 'L' or
(m) if *side* = 'R'.
work is not referenced if *H* has order < 11.

Output Parameters

c On exit, *C* is overwritten by the matrix *H*C* if *side* = 'L', or *C*H* if *side* = 'R'.

?largv

Generates a vector of plane rotations with real cosines and real/complex sines.

Syntax

```
call slargv( n, x, incx, y, incy, c, incc )
call dlargv( n, x, incx, y, incy, c, incc )
call clargv( n, x, incx, y, incy, c, incc )
call zlargv( n, x, incx, y, incy, c, incc )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine generates a vector of real/complex plane rotations with real cosines, determined by elements of the real/complex vectors *x* and *y*.

For slargv/dlargv:

$$\begin{bmatrix} c(i) & s(i) \\ -s(i) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} a_i \\ 0 \end{bmatrix}, \text{ for } i = 1, 2, \dots, n$$

For clargv/zlargv:

$$\begin{bmatrix} c(i) & s(i) \\ -\text{conj}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} r_i \\ 0 \end{bmatrix}, \text{ for } i = 1, 2, \dots, n$$

where $c(i)^2 + \text{abs}(s(i))^2 = 1$ and the following conventions are used (these are the same as in clartg/zlartg but differ from the BLAS Level 1 routine crotg/zrotg):

If $y_i = 0$, then $c(i) = 1$ and $s(i) = 0$;

If $x_i = 0$, then $c(i) = 0$ and $s(i)$ is chosen so that r_i is real.

Input Parameters

n INTEGER. The number of plane rotations to be generated.

x, y REAL for slargv

DOUBLE PRECISION for dlargv

COMPLEX for clargv

DOUBLE COMPLEX for zlargv

Arrays, DIMENSION (1+(n-1)*incx) and (1+(n-1)*incy), respectively. On entry, the vectors *x* and *y*.

incx INTEGER. The increment between elements of *x*.

incx > 0.

incy INTEGER. The increment between elements of *y*.

incy > 0.

incc INTEGER. The increment between elements of the output array *c*. *incc* > 0.

Output Parameters

x On exit, *x*(*i*) is overwritten by *a*_{*i*} (for real flavors), or by *r*_{*i*} (for complex flavors), for *i* = 1, ..., *n*.

y On exit, the sines *s*(*i*) of the plane rotations.

c REAL for slargv/clargv

DOUBLE PRECISION for dlargv/zlargv

Array, DIMENSION (1+(n-1)*incc). The cosines of the plane rotations.

?larnv

Returns a vector of random numbers from a uniform or normal distribution.

Syntax

call slarnv(*idist*, *iseed*, *n*, *x*)

```
call dlarnv( idist, iseed, n, x )
call clarnv( idist, iseed, n, x )
call zlarnv( idist, iseed, n, x )
```

C:

```
lapack_int LAPACKE_<?>larnv (lapack_int idist, lapack_int * iseed, lapack_int n,
<datatype> * x);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?larnv returns a vector of *n* random real/complex numbers from a uniform or normal distribution.

This routine calls the auxiliary routine ?laruv to generate random real numbers from a uniform (0,1) distribution, in batches of up to 128 using vectorisable code. The Box-Muller method is used to transform numbers from a uniform to a normal distribution.

Input Parameters

The data types are given for the Fortran interface.

<i>idist</i>	INTEGER. Specifies the distribution of the random numbers: for slarnv and dlanrv: = 1: uniform (0,1) = 2: uniform (-1,1) = 3: normal (0,1). for clar nv and zlanrv: = 1: real and imaginary parts each uniform (0,1) = 2: real and imaginary parts each uniform (-1,1) = 3: real and imaginary parts each normal (0,1) = 4: uniformly distributed on the disc abs(z) < 1 = 5: uniformly distributed on the circle abs(z) = 1
<i>iseed</i>	INTEGER. Array, DIMENSION (4). On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and <i>iseed</i> (4) must be odd.
<i>n</i>	INTEGER. The number of random numbers to be generated.

Output Parameters

<i>x</i>	REAL for slarnv DOUBLE PRECISION for dlanrv COMPLEX for clar nv DOUBLE COMPLEX for zlanrv
----------	--

Array, DIMENSION (n). The generated random numbers.

iseed On exit, the seed is updated.

?larra

Computes the splitting points with the specified threshold.

Syntax

```
call slarra( n, d, e, e2, spltol, tnrm, nspli, isplit, info )
call dlarra( n, d, e, e2, spltol, tnrm, nspli, isplit, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the splitting points with the specified threshold and sets any "small" off-diagonal elements to zero.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix ($n > 1$).
<i>d</i>	REAL for slarra DOUBLE PRECISION for dlarra Array, DIMENSION (n). Contains n diagonal elements of the tridiagonal matrix T .
<i>e</i>	REAL for slarra DOUBLE PRECISION for dlarra Array, DIMENSION (n). First $(n-1)$ entries contain the subdiagonal elements of the tridiagonal matrix T ; $e(n)$ need not be set.
<i>e2</i>	REAL for slarra DOUBLE PRECISION for dlarra Array, DIMENSION (n). First $(n-1)$ entries contain the squares of the subdiagonal elements of the tridiagonal matrix T ; $e2(n)$ need not be set.
<i>spltol</i>	REAL for slarra DOUBLE PRECISION for dlarra

The threshold for splitting. Two criteria can be used: $spltol < 0$: criterion based on absolute off-diagonal value; $spltol > 0$: criterion that preserves relative accuracy.

tnrm REAL **for** slarra
 DOUBLE PRECISION **for** dlarra
 The norm of the matrix.

Output Parameters

e On exit, the entries $e(isplit(i))$, $1 \leq i \leq nsplit$, are set to zero, the other entries of *e* are untouched.

e2 On exit, the entries $e2(isplit(i))$, $1 \leq i \leq nsplit$, are set to zero.

nsplit INTEGER.
 The number of blocks the matrix *T* splits into. $1 \leq nsplit \leq n$

isplit INTEGER.
 Array, DIMENSION (*n*).
 The splitting points, at which *T* breaks up into blocks. The first block consists of rows/columns 1 to *isplit*(1), the second of rows/columns *isplit*(1)+1 through *isplit*(2), and so on, and the *nsplit*-th consists of rows/columns *isplit*(*nsplit*-1)+1 through *isplit*(*nsplit*)=*n*.

info INTEGER.
 = 0: successful exit.

?larrb

Provides limited bisection to locate eigenvalues for more accuracy.

Syntax

```
call slarrb( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work,  

iwork, pivmin, spdiam, twist, info )  

call dlarrb( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work,  

iwork, pivmin, spdiam, twist, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

Given the relatively robust representation (RRR) $L^*D^*L^T$, the routine does "limited" bisection to refine the eigenvalues of $L^*D^*L^T$, w (*ifirst-offset*) through w (*ilast-offset*), to more accuracy. Initial guesses for these eigenvalues are input in *w*. The corresponding estimate of the error in these guesses and their gaps are input in *werr* and *wgap*, respectively. During bisection, intervals $[left, right]$ are maintained by storing their midpoints and semi-widths in the arrays *w* and *werr* respectively.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix.
<i>d</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (<i>n</i>). The <i>n</i> diagonal elements of the diagonal matrix D .
<i>lld</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (<i>n-1</i>). The <i>n-1</i> elements $L_i^* L_i^* D_i$.
<i>ifirst</i>	INTEGER. The index of the first eigenvalue to be computed.
<i>ilast</i>	INTEGER. The index of the last eigenvalue to be computed.
<i>rto11, rto12</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Tolerance for the convergence of the bisection intervals. An interval [<i>left</i> , <i>right</i>] has converged if $\text{RIGHT-LEFT.LT.MAX}(\ rto11*\text{gap},\ rto12*\max(\text{left} , \text{right}))$, where <i>gap</i> is the (estimated) distance to the nearest eigenvalue.
<i>offset</i>	INTEGER. Offset for the arrays <i>w</i> , <i>wgap</i> and <i>werr</i> , that is, the <i>ifirst-offset</i> through <i>ilast-offset</i> elements of these arrays are to be used.
<i>w</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (<i>n</i>). On input, <i>w</i> (<i>ifirst-offset</i>) through <i>w</i> (<i>ilast-offset</i>) are estimates of the eigenvalues of $L^*D^*L^T$ indexed <i>ifirst</i> through <i>ilast</i> .
<i>wgap</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (<i>n-1</i>). The estimated gaps between consecutive eigenvalues of $L^*D^*L^T$, that is, <i>wgap</i> (<i>i-offset</i>) is the gap between eigenvalues <i>i</i> and <i>i+1</i> . Note that if <i>IFIRST.EQ.ILAST</i> then <i>wgap</i> (<i>ifirst-offset</i>) must be set to 0.
<i>werr</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Array, DIMENSION (<i>n</i>). On input, <i>werr</i> (<i>ifirst-offset</i>) through <i>werr</i> (<i>ilast-offset</i>) are the errors in the estimates of the corresponding elements in <i>w</i> .
<i>work</i>	REAL for slarrb DOUBLE PRECISION for dlarrb Workspace array, DIMENSION ($2*n$).

<i>pivmin</i>	REAL for slarrb DOUBLE PRECISION for dlarrb The minimum pivot in the Sturm sequence.
<i>spdiam</i>	REAL for slarrb DOUBLE PRECISION for dlarrb The spectral diameter of the matrix.
<i>twist</i>	INTEGER. The twist index for the twisted factorization that is used for the negcount. $twist = n$: Compute negcount from $L^* D^* L^T - \lambda * i = L^* D + * L^T$ $twist = n$: Compute negcount from $L^* D^* L^T - \lambda * i = U^* D^- * U^{-T}$ $twist = n$: Compute negcount from $L^* D^* L^T - \lambda * i = N_r^* D^- * N_r$
<i>iwork</i>	INTEGER. Workspace array, DIMENSION (2*n).

Output Parameters

<i>w</i>	On output, the estimates of the eigenvalues are "refined".
<i>wgap</i>	On output, the gaps are refined.
<i>werr</i>	On output, "refined" errors in the estimates of <i>w</i> .
<i>info</i>	INTEGER. Error flag.

?larrc

Computes the number of eigenvalues of the symmetric tridiagonal matrix.

Syntax

```
call slarrc( jobt, n, vl, vu, d, e, pivmin, eigcnt, lcnt, rcnt, info )
call dlarrc( jobt, n, vl, vu, d, e, pivmin, eigcnt, lcnt, rcnt, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine finds the number of eigenvalues of the symmetric tridiagonal matrix T or of its factorization $L^* D^* L^T$ in the specified interval.

Input Parameters

<i>jobt</i>	CHARACTER*1. = 'T': computes Sturm count for matrix T . = 'L': computes Sturm count for matrix $L^*D^*L^T$.
<i>n</i>	INTEGER. The order of the matrix. ($n > 1$).
<i>vl,vu</i>	REAL for slarrc DOUBLE PRECISION for dlarrc The lower and upper bounds for the eigenvalues.
<i>d</i>	REAL for slarrc DOUBLE PRECISION for dlarrc Array, DIMENSION (n). If <i>jobt</i> = 'T': contains the n diagonal elements of the tridiagonal matrix T . If <i>jobt</i> = 'L': contains the n diagonal elements of the diagonal matrix D .
<i>e</i>	REAL for slarrc DOUBLE PRECISION for dlarrc Array, DIMENSION (n). If <i>jobt</i> = 'T': contains the $(n-1)$ offdiagonal elements of the matrix T . If <i>jobt</i> = 'L': contains the $(n-1)$ offdiagonal elements of the matrix L .
<i>pivmin</i>	REAL for slarrc DOUBLE PRECISION for dlarrc The minimum pivot in the Sturm sequence for the matrix T .

Output Parameters

<i>eigcnt</i>	INTEGER. The number of eigenvalues of the symmetric tridiagonal matrix T that are in the half-open interval $(vl, vu]$.
<i>lcnt,rcnt</i>	INTEGER. The left and right negcounts of the interval.
<i>info</i>	INTEGER. Now it is not used and always is set to 0.

?larrd

Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.

Syntax

```
call slarrd( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplitt,
isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, info )
```

```
call dlarrd( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplitt,
isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the eigenvalues of a symmetric tridiagonal matrix T to suitable accuracy. This is an auxiliary code to be called from ?stemr. The user may ask for all eigenvalues, all eigenvalues in the half-open interval $(vl, vu]$, or the il -th through iu -th eigenvalues.

To avoid overflow, the matrix must be scaled so that its largest element is no greater than $(\text{overflow}^{1/2} * \text{underflow}^{1/4})$ in absolute value, and for greatest accuracy, it should not be much smaller than that. (For more details see [Kahan66].

Input Parameters

<i>range</i>	CHARACTER. = 'A': ("All") all eigenvalues will be found. = 'V': ("Value") all eigenvalues in the half-open interval $(vl, vu]$ will be found. = 'I': ("Index") the il -th through iu -th eigenvalues will be found.
<i>order</i>	CHARACTER. = 'B': ("By block") the eigenvalues will be grouped by split-off block (see <i>iblock</i> , <i>isplit</i> below) and ordered from smallest to largest within the block. = 'E': ("Entire matrix") the eigenvalues for the entire matrix will be ordered from smallest to largest.
<i>n</i>	INTEGER. The order of the tridiagonal matrix T ($n \geq 1$).
<i>vl,vu</i>	REAL for slarrd DOUBLE PRECISION for dlarrd If <i>range</i> = 'V': the lower and upper bounds of the interval to be searched for eigenvalues. Eigenvalues less than or equal to <i>vl</i> , or greater than <i>vu</i> , will not be returned. <i>vl</i> < <i>vu</i> . If <i>range</i> = 'A' or 'I': not referenced.
<i>il,iu</i>	INTEGER. If <i>range</i> = 'I': the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq il \leq iu \leq n$, if $n > 0$; <i>il</i> =1 and <i>iu</i> =0 if $n=0$. If <i>range</i> = 'A' or 'V': not referenced.

<i>gers</i>	REAL for slarrd DOUBLE PRECISION for dlarrd Array, DIMENSION ($2 \times n$). The n Gershgorin intervals (the i -th Gershgorin interval is ($gers(2*i-1)$, $gers(2*i)$)).
<i>reltol</i>	REAL for slarrd DOUBLE PRECISION for dlarrd The minimum relative width of an interval. When an interval is narrower than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, that is converged. Note: this should always be at least <i>radix*machine epsilon</i> .
<i>d</i>	REAL for slarrd DOUBLE PRECISION for dlarrd Array, DIMENSION (n). Contains n diagonal elements of the tridiagonal matrix T .
<i>e</i>	REAL for slarrd DOUBLE PRECISION for dlarrd Array, DIMENSION ($n-1$). Contains $(n-1)$ off-diagonal elements of the tridiagonal matrix T .
<i>e2</i>	REAL for slarrd DOUBLE PRECISION for dlarrd Array, DIMENSION ($n-1$). Contains $(n-1)$ squared off-diagonal elements of the tridiagonal matrix T .
<i>pivmin</i>	REAL for slarrd DOUBLE PRECISION for dlarrd The minimum pivot in the Sturm sequence for the matrix T .
<i>nsplit</i>	INTEGER. The number of diagonal blocks the matrix T . $1 \leq nsplit \leq n$
<i>isplit</i>	INTEGER. Arrays, DIMENSION (n). The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to <i>isplit</i> (1), the second of rows/columns <i>isplit</i> (1)+1 through <i>isplit</i> (2), and so on, and the <i>nsplit</i> -th consists of rows/columns <i>isplit</i> (<i>nsplit</i> -1)+1 through <i>isplit</i> (<i>nsplit</i>)= n . (Only the first <i>nsplit</i> elements actually is used, but since the user cannot know a priori value of <i>nsplit</i> , n words must be reserved for <i>isplit</i> .)

work REAL for slarrd
 DOUBLE PRECISION for dlarrd
 Workspace array, DIMENSION (4*n).

iwork INTEGER.
 Workspace array, DIMENSION (4*n).

Output Parameters

m INTEGER.
 The actual number of eigenvalues found. $0 \leq m \leq n$. (See also the description of *info=2, 3*.)

w REAL for slarrd
 DOUBLE PRECISION for dlarrd
 Array, DIMENSION (n).
 The first *m* elements of *w* contain the eigenvalue approximations. ?laprd computes an interval $I_j = (a_j, b_j]$ that includes eigenvalue *j*. The eigenvalue approximation is given as the interval midpoint $w(j) = (a_j + b_j)/2$. The corresponding error is bounded by $werr(j) = \text{abs}(a_j - b_j)/2$.

werr REAL for slarrd
 DOUBLE PRECISION for dlarrd
 Array, DIMENSION (n).
 The error bound on the corresponding eigenvalue approximation in *w*.

wl, wu REAL for slarrd
 DOUBLE PRECISION for dlarrd
 The interval $(wl, wu]$ contains all the wanted eigenvalues.
If *range* = 'V': then *wl*=*vl* and *wu*=*vu*.
If *range* = 'A': then *wl* and *wu* are the global Gerschgorin bounds on the spectrum.
If *range* = 'I': then *wl* and *wu* are computed by ?laebz from the index range specified.

iblock INTEGER.
 Array, DIMENSION (n).
At each row/column *j* where $e(j)$ is zero or small, the matrix *T* is considered to split into a block diagonal matrix.
If *info* = 0, then *iblock(i)* specifies to which block (from 1 to the number of blocks) the eigenvalue *w(i)* belongs. (The routine may use the remaining $n-m$ elements as workspace.)

indexw INTEGER.
 Array, DIMENSION (n).

The indices of the eigenvalues within each block (submatrix); for example, $indexw(i) = j$ and $iblock(i)=k$ imply that the i -th eigenvalue $w(i)$ is the j -th eigenvalue in block k .

info

INTEGER.

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument has an illegal value

> 0: some or all of the eigenvalues fail to converge or are not computed:

=1 or 3: bisection fail to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.

=2 or 3: *range*='I' only: not all of the eigenvalues *il*:*iu* are found.=4: *range*='I', and the Gershgorin interval initially used is too small. No eigenvalues are computed.

?larre

Given the tridiagonal matrix T , sets small off-diagonal elements to zero and for each unreduced block T_i , finds base representations and eigenvalues.

Syntax

```
call slarre( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit, isplit,
m, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
call dlarre( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit, isplit,
m, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T , the routine sets any "small" off-diagonal elements to zero, and for each unreduced block T_i , it finds

- a suitable shift at one end of the block spectrum
- the base representation, $T_i - \sigma_i * I = L_i * D_i * L_i^T$, and
- eigenvalues of each $L_i * D_i * L_i^T$.

The representations and eigenvalues found are then used by ?stemr to compute the eigenvectors of a symmetric tridiagonal matrix. The accuracy varies depending on whether bisection is used to find a few eigenvalues or the dqds algorithm (subroutine ?lasq2) to compute all and discard any unwanted one. As an added benefit, ?larre also outputs the n Gerschgorin intervals for the matrices $L_i * D_i * L_i^T$.

Input Parameters

range

CHARACTER.

= 'A': ("All") all eigenvalues will be found.

= 'V': ("Value") all eigenvalues in the half-open interval (vl , vu] will be found.

= 'I': ("Index") the il -th through iu -th eigenvalues of the entire matrix will be found.

n INTEGER. The order of the matrix. $n > 0$.

vl , vu REAL for slarre

DOUBLE PRECISION for dlarre

If $range='V'$, the lower and upper bounds for the eigenvalues.

Eigenvalues less than or equal to vl , or greater than vu , are not returned.
 $vl < vu$.

il , iu INTEGER.

If $range='I'$, the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq il \leq iu \leq n$.

d REAL for slarre

DOUBLE PRECISION for dlarre

Array, DIMENSION (n).

The n diagonal elements of the diagonal matrices T .

e REAL for slarre

DOUBLE PRECISION for dlarre

Array, DIMENSION (n). The first ($n-1$) entries contain the subdiagonal elements of the tridiagonal matrix T ; $e(n)$ need not be set.

$e2$ REAL for slarre

DOUBLE PRECISION for dlarre

Array, DIMENSION (n). The first ($n-1$) entries contain the squares of the subdiagonal elements of the tridiagonal matrix T ; $e2(n)$ need not be set.

$rto11$, $rto12$ REAL for slarre

DOUBLE PRECISION for dlarre

Parameters for bisection. An interval [LEFT,RIGHT] has converged if
 $RIGHT-LEFT.LT.MAX(rto11*gap, rto12*\max(|LEFT|,|RIGHT|))$.

$spltol$ REAL for slarre

DOUBLE PRECISION for dlarre

The threshold for splitting.

$work$ REAL for slarre

DOUBLE PRECISION for dlarre

Workspace array, DIMENSION ($6*n$).

$iwork$ INTEGER.

Workspace array, DIMENSION (5*n).

Output Parameters

<i>vl</i> , <i>vu</i>	On exit, if <i>range</i> ='I' or ='A', contain the bounds on the desired part of the spectrum.
<i>d</i>	On exit, the <i>n</i> diagonal elements of the diagonal matrices D_i .
<i>e</i>	On exit, the subdiagonal elements of the unit bidiagonal matrices L_i . The entries <i>e</i> (<i>isplit</i> (<i>i</i>)), $1 \leq i \leq nsplit$, contain the base points <i>sigma</i> _{<i>i</i>} on output.
<i>e2</i>	On exit, the entries <i>e2</i> (<i>isplit</i> (<i>i</i>)), $1 \leq i \leq nsplit$, have been set to zero.
<i>nsplit</i>	INTEGER. The number of blocks <i>T</i> splits into. $1 \leq nsplit \leq n$.
<i>isplit</i>	INTEGER. Array, DIMENSION (<i>n</i>). The splitting points, at which <i>T</i> breaks up into blocks. The first block consists of rows/columns 1 to <i>isplit</i> (1), the second of rows/columns <i>isplit</i> (1)+1 through <i>isplit</i> (2), etc., and the <i>nsplit</i> -th consists of rows/columns <i>isplit</i> (<i>nsplit</i> -1)+1 through <i>isplit</i> (<i>nsplit</i>)= <i>n</i> .
<i>m</i>	INTEGER. The total number of eigenvalues (of all the $L_i * D_i * L_i^T$) found.
<i>w</i>	REAL for slarre DOUBLE PRECISION for dlarre Array, DIMENSION (<i>n</i>). The first <i>m</i> elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i * D_i * L_i^T$, are sorted in ascending order. The routine may use the remaining <i>n-m</i> elements as workspace.
<i>werr</i>	REAL for slarre DOUBLE PRECISION for dlarre Array, DIMENSION (<i>n</i>). The error bound on the corresponding eigenvalue in <i>w</i> .
<i>wgap</i>	REAL for slarre DOUBLE PRECISION for dlarre Array, DIMENSION (<i>n</i>). The separation from the right neighbor eigenvalue in <i>w</i> . The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree. Exception: at the right end of a block the left gap is stored.
<i>iblock</i>	INTEGER. Array, DIMENSION (<i>n</i>). The indices of the blocks (submatrices) associated with the corresponding eigenvalues in <i>w</i> ; <i>iblock</i> (<i>i</i>)=1 if eigenvalue <i>w</i> (<i>i</i>) belongs to the first block from the top, =2 if <i>w</i> (<i>i</i>) belongs to the second block, etc.
<i>indexw</i>	INTEGER. Array, DIMENSION (<i>n</i>).

The indices of the eigenvalues within each block (submatrix); for example, $\text{indexw}(i)=10$ and $\text{iblock}(i)=2$ imply that the i -th eigenvalue $w(i)$ is the 10-th eigenvalue in the second block.

gers

REAL for slarre

DOUBLE PRECISION for dlarre

Array, DIMENSION ($2*n$). The n Gerschgorin intervals (the i -th Gerschgorin interval is $(\text{gers}(2*i-1), \text{gers}(2*i))$).

pivmin

REAL for slarre

DOUBLE PRECISION for dlarre

The minimum pivot in the Sturm sequence for T .

info

INTEGER.

If $\text{info} = 0$: successful exit

If $\text{info} > 0$: A problem occurred in ?larre. If $\text{info} = 5$, the Rayleigh Quotient Iteration failed to converge to full accuracy.

If $\text{info} < 0$: One of the called subroutines signaled an internal problem. Inspection of the corresponding parameter info for further information is required.

- If $\text{info} = -1$, there is a problem in ?larrd
- If $\text{info} = -2$, no base representation could be found in `maxtry` iterations. Increasing `maxtry` and recompilation might be a remedy.
- If $\text{info} = -3$, there is a problem in ?larrb when computing the refined root representation for ?lasq2.
- If $\text{info} = -4$, there is a problem in ?larrb when performing bisection on the desired part of the spectrum.
- If $\text{info} = -5$, there is a problem in ?lasq2.
- If $\text{info} = -6$, there is a problem in ?lasq2.

See Also

[?stemr](#)[?lasq2](#)[?larrb](#)[?larrd](#)

?larrf

Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.

Syntax

```
call slarrf( n, d, l, ld, clstrt, clenld, w, wgap, werr, spdiam, clgap1, clgapr,
pivmin, sigma, dplus, lplus, work, info )
```

```
call dlarrf( n, d, l, ld, clstrt, clenld, w, wgap, werr, spdiam, clgap1, clgapr,
pivmin, sigma, dplus, lplus, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

Given the initial representation L^*D*L^T and its cluster of close eigenvalues (in a relative measure), $w(clstrt)$, $w(clstrt+1)$, ... $w(clend)$, the routine ?larrf finds a new relatively robust representation

$$L^*D*L^T - \sigma_i * I = L(+) * D(+) * L(+)^T$$

such that at least one of the eigenvalues of $L(+)*D(+)*L(+)^T$ is relatively isolated.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix (subblock, if the matrix is splitted).
<i>d</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (<i>n</i>). The <i>n</i> diagonal elements of the diagonal matrix <i>D</i> .
<i>l</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (<i>n</i> -1). The (<i>n</i> -1) subdiagonal elements of the unit bidiagonal matrix <i>L</i> .
<i>ld</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (<i>n</i> -1). The <i>n</i> -1 elements L_i*D_i .
<i>clstrt</i>	INTEGER. The index of the first eigenvalue in the cluster.
<i>clend</i>	INTEGER. The index of the last eigenvalue in the cluster.
<i>w</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION $\geq (clend - clstrt + 1)$. The eigenvalue approximations of L^*D*L^T in ascending order. $w(clstrt)$ through $w(clend)$ form the cluster of relatively close eigenvalues.
<i>wgap</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION $\geq (clend - clstrt + 1)$. The separation from the right neighbor eigenvalue in <i>w</i> .
<i>werr</i>	REAL for slarrf DOUBLE PRECISION for dlarrf

Array, DIMENSION $\geq (clend - clstrt + 1)$. On input, *werr* contains the semiwidth of the uncertainty interval of the corresponding eigenvalue approximation in *w*.

<i>spdiam</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Estimate of the spectral diameter obtained from the Gerschgorin intervals.
<i>clgapl, clgapr</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Absolute gap on each end of the cluster. Set by the calling routine to protect against shifts too close to eigenvalues outside the cluster.
<i>pivmin</i>	REAL for slarrf DOUBLE PRECISION for dlarrf The minimum pivot allowed in the Sturm sequence.
<i>work</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Workspace array, DIMENSION $(2*n)$.

Output Parameters

<i>wgap</i>	On output, the gaps are refined.
<i>sigma</i>	REAL for slarrf DOUBLE PRECISION for dlarrf The shift used to form $L(+)^*D^*(+)^*L(+)^T$.
<i>dplus</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (n) . The n diagonal elements of the diagonal matrix $D(+)$.
<i>lplus</i>	REAL for slarrf DOUBLE PRECISION for dlarrf Array, DIMENSION (n) . The first $(n-1)$ elements of <i>lplus</i> contain the subdiagonal elements of the unit bidiagonal matrix $L(+)^*$.

?larrj

Performs refinement of the initial estimates of the eigenvalues of the matrix *T*.

Syntax

```
call slarrj( n, d, e2, ifirst, ilast, rtol, offset, w, werr, work, iwork,
pivmin, spdiam, info )
```

```
call dlarrj( n, d, e2, ifirst, ilast, rtol, offset, w, werr, work, iwork,
pivmin, spdiam, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

Given the initial eigenvalue approximations of T , this routine does bisection to refine the eigenvalues of T , $w(ifirst-offset)$ through $w(ilast-offset)$, to more accuracy. Initial guesses for these eigenvalues are input in w , the corresponding estimate of the error in these guesses in $werr$. During bisection, intervals $[a,b]$ are maintained by storing their mid-points and semi-widths in the arrays w and $werr$ respectively.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix T .
<i>d</i>	<p>REAL for slarrj</p> <p>DOUBLE PRECISION for dlarrj</p> <p>Array, DIMENSION (<i>n</i>). Contains <i>n</i> diagonal elements of the matrix T.</p>
<i>e2</i>	<p>REAL for slarrj</p> <p>DOUBLE PRECISION for dlarrj</p> <p>Array, DIMENSION (<i>n-1</i>). Contains (<i>n-1</i>) squared sub-diagonal elements of the T.</p>
<i>ifirst</i>	INTEGER. The index of the first eigenvalue to be computed.
<i>ilast</i>	INTEGER. The index of the last eigenvalue to be computed.
<i>rtol</i>	<p>REAL for slarrj</p> <p>DOUBLE PRECISION for dlarrj</p> <p>Tolerance for the convergence of the bisection intervals. An interval $[a,b]$ is considered to be converged if $(b-a) \leq rtol * \max(a , b)$.</p>
<i>offset</i>	INTEGER. Offset for the arrays w and $werr$, that is the $ifirst-offset$ through $ilast-offset$ elements of these arrays are to be used.
<i>w</i>	<p>REAL for slarrj</p> <p>DOUBLE PRECISION for dlarrj</p> <p>Array, DIMENSION (<i>n</i>). On input, $w(ifirst-offset)$ through $w(ilast-offset)$ are estimates of the eigenvalues of $L^*D^*L^T$ indexed <i>ifirst</i> through <i>ilast</i>.</p>

<i>werr</i>	REAL for slarrj DOUBLE PRECISION for dlarrj Array, DIMENSION (<i>n</i>). On input, <i>werr</i> (<i>ifirst-offset</i>) through <i>werr</i> (<i>ilast-offset</i>) are the errors in the estimates of the corresponding elements in <i>w</i> .
<i>work</i>	REAL for slarrj DOUBLE PRECISION for dlarrj Workspace array, DIMENSION (2*n).
<i>iwork</i>	INTEGER. Workspace array, DIMENSION (2*n).
<i>pivmin</i>	REAL for slarrj DOUBLE PRECISION for dlarrj The minimum pivot in the Sturm sequence for the matrix <i>T</i> .
<i>spdiam</i>	REAL for slarrj DOUBLE PRECISION for dlarrj The spectral diameter of the matrix <i>T</i> .

Output Parameters

<i>w</i>	On exit, contains the refined estimates of the eigenvalues.
<i>werr</i>	On exit, contains the refined errors in the estimates of the corresponding elements in <i>w</i> .
<i>info</i>	INTEGER. Now it is not used and always is set to 0.

?larrk

Computes one eigenvalue of a symmetric tridiagonal matrix *T* to suitable accuracy.

Syntax

```
call slarrk( n, iw, gl, gu, d, e2, pivmin, reltol, w, werr, info )
call dlarrk( n, iw, gl, gu, d, e2, pivmin, reltol, w, werr, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes one eigenvalue of a symmetric tridiagonal matrix *T* to suitable accuracy. This is an auxiliary code to be called from ?stemr.

To avoid overflow, the matrix must be scaled so that its largest element is no greater than $(\text{overflow}^{1/2} * \text{underflow}^{1/4})$ in absolute value, and for greatest accuracy, it should not be much smaller than that. For more details see [Kahan66].

Input Parameters

<i>n</i>	INTEGER. The order of the matrix T . ($n \geq 1$).
<i>iw</i>	INTEGER. The index of the eigenvalue to be returned.
<i>gl, gu</i>	REAL for slarrk DOUBLE PRECISION for dlarrk An upper and a lower bound on the eigenvalue.
<i>d</i>	REAL for slarrk DOUBLE PRECISION for dlarrk Array, DIMENSION (n). Contains n diagonal elements of the matrix T .
<i>e2</i>	REAL for slarrk DOUBLE PRECISION for dlarrk Array, DIMENSION ($n-1$). Contains $(n-1)$ squared off-diagonal elements of the T .
<i>pivmin</i>	REAL for slarrk DOUBLE PRECISION for dlarrk The minimum pivot in the Sturm sequence for the matrix T .
<i>reltol</i>	REAL for slarrk DOUBLE PRECISION for dlarrk The minimum relative width of an interval. When an interval is narrower than <i>reltol</i> times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, that is converged. Note: this should always be at least <i>radix*machine epsilon</i> .

Output Parameters

<i>w</i>	REAL for slarrk DOUBLE PRECISION for dlarrk Contains the eigenvalue approximation.
<i>werr</i>	REAL for slarrk DOUBLE PRECISION for dlarrk Contains the error bound on the corresponding eigenvalue approximation in <i>w</i> .

<i>info</i>	INTEGER. = 0: Eigenvalue converges = -1: Eigenvalue does not converge
-------------	---

?larrr

Performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.

Syntax

```
call slarrr( n, d, e, info )
call dlarrr( n, d, e, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine performs tests to decide whether the symmetric tridiagonal matrix T warrants expensive computations which guarantee high relative accuracy in the eigenvalues.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix T . ($n > 0$).
<i>d</i>	REAL for slarrr DOUBLE PRECISION for dlarrr Array, DIMENSION (<i>n</i>). Contains <i>n</i> diagonal elements of the matrix T .
<i>e</i>	REAL for slarrr DOUBLE PRECISION for dlarrr Array, DIMENSION (<i>n</i>). The first (<i>n</i> -1) entries contain sub-diagonal elements of the tridiagonal matrix T ; <i>e</i> (<i>n</i>) is set to 0.

Output Parameters

<i>info</i>	INTEGER. = 0: the matrix warrants computations preserving relative accuracy (default value). = -1: the matrix warrants computations guaranteeing only absolute accuracy.
-------------	--

?larrv

*Computes the eigenvectors of the tridiagonal matrix $T = L^*D^*L^T$ given L , D and the eigenvalues of $L^*D^*L^T$.*

Syntax

```
call slarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtoll, rtol2, w,
werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
call dlarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtoll, rtol2, w,
werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
call clarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtoll, rtol2, w,
werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
call zlarrv( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, minrgp, rtoll, rtol2, w,
werr, wgap, iblock, indexw, gers, z, ldz, isuppz, work, iwork, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?larrv computes the eigenvectors of the tridiagonal matrix $T = L^*D^*L^T$ given L , D and approximations to the eigenvalues of $L^*D^*L^T$.

The input eigenvalues should have been computed by slarre for real flavors (slarrv/clarrv) and by dlarre for double precision flavors (dlarre/zlarre).

Input Parameters

<i>n</i>	INTEGER. The order of the matrix. $n \geq 0$.
<i>vl</i> , <i>vu</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Lower and upper bounds respectively of the interval that contains the desired eigenvalues. $vl < vu$. Needed to compute gaps on the left or right end of the extremal eigenvalues in the desired range.
<i>d</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Array, DIMENSION (<i>n</i>). On entry, the <i>n</i> diagonal elements of the diagonal matrix D .
<i>l</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Array, DIMENSION (<i>n</i>). On entry, the (<i>n</i> -1) subdiagonal elements of the unit bidiagonal matrix L are contained in elements 1 to <i>n</i> -1 of <i>L</i> if the matrix is not splitted. At the end of each block the corresponding shift is stored as given by slarre for real flavors and by dlarre for double precision flavors.

<i>pivmin</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv The minimum pivot allowed in the Sturm sequence.
<i>isplit</i>	INTEGER. Array, DIMENSION (<i>n</i>). The splitting points, at which <i>T</i> breaks up into blocks. The first block consists of rows/columns 1 to <i>isplit</i> (1), the second of rows/columns <i>isplit</i> (1)+1 through <i>isplit</i> (2), etc.
<i>m</i>	INTEGER. The total number of eigenvalues found. $0 \leq m \leq n$. If <i>range</i> = 'A', <i>m</i> = <i>n</i> , and if <i>range</i> = 'I', <i>m</i> = <i>iu</i> - <i>il</i> +1.
<i>dol, dou</i>	INTEGER. If you want to compute only selected eigenvectors from all the eigenvalues supplied, specify an index range <i>dol:dou</i> . Or else apply the setting <i>dol</i> =1, <i>dou</i> = <i>m</i> . Note that <i>dol</i> and <i>dou</i> refer to the order in which the eigenvalues are stored in <i>w</i> . If you want to compute only selected eigenpairs, then the columns <i>dol</i> -1 to <i>dou</i> +1 of the eigenvector space <i>Z</i> contain the computed eigenvectors. All other columns of <i>Z</i> are set to zero.
<i>minrgp, rtol1, rtol2</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Parameters for bisection. An interval [LEFT,RIGHT] has converged if RIGHT-LEFT.LT.MAX(<i>rtol1</i> *gap, <i>rtol2</i> *max(LEFT , RIGHT)).
<i>w</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Array, DIMENSION (<i>n</i>). The first <i>m</i> elements of <i>w</i> contain the approximate eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block (the output array <i>w</i> from ?larre is expected here). These eigenvalues are set with respect to the shift of the corresponding root representation for their block.
<i>werr</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Array, DIMENSION (<i>n</i>). The first <i>m</i> elements contain the semiwidth of the uncertainty interval of the corresponding eigenvalue in <i>w</i> .
<i>wgap</i>	REAL for slarrv/clarrv DOUBLE PRECISION for dlarrv/zlarrv Array, DIMENSION (<i>n</i>). The separation from the right neighbor eigenvalue in <i>w</i> .
<i>iblock</i>	INTEGER. Array, DIMENSION (<i>n</i>).

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in w ; $iblock(i)=1$ if eigenvalue $w(i)$ belongs to the first block from the top, $=2$ if $w(i)$ belongs to the second block, etc.

*indexw*INTEGER. Array, DIMENSION (n).

The indices of the eigenvalues within each block (submatrix); for example, $indexw(i)=10$ and $iblock(i)=2$ imply that the i -th eigenvalue $w(i)$ is the 10-th eigenvalue in the second block.

gers

REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

Array, DIMENSION ($2*n$). The n Gerschgorin intervals (the i -th Gerschgorin interval is $(gers(2*i-1), gers(2*i))$). The Gerschgorin intervals should be computed from the original unshifted matrix.

ldz

INTEGER. The leading dimension of the output array Z . $ldz \geq 1$, and if $jobz = 'V'$, $ldz \geq \max(1, n)$.

work

REAL for slarrv/clarrv

DOUBLE PRECISION for dlarrv/zlarrv

Workspace array, DIMENSION ($12*n$).*iwork*

INTEGER.

Workspace array, DIMENSION ($7*n$).

Output Parameters

*d*On exit, d may be overwritten.*l*On exit, l is overwritten.*w*On exit, w holds the eigenvalues of the unshifted matrix.*werr*On exit, $werr$ contains refined values of its input approximations.*wgap*

On exit, $wgap$ contains refined values of its input approximations. Very small gaps are changed.

z

REAL for slarrv

DOUBLE PRECISION for dlarrv

COMPLEX for clarrv

DOUBLE COMPLEX for zlarrv

Array, DIMENSION ($ldz, \max(1,m)$).

If $info = 0$, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the input eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

NOTE

The user must ensure that at least $\max(1,m)$ columns are supplied in the array z .

isuppz

INTEGER .

Array, DIMENSION ($2 * \max(1, m)$). The support of the eigenvectors in z , that is, the indices indicating the nonzero elements in z . The i -th eigenvector is nonzero only in elements $isuppz(2i-1)$ through $isuppz(2i)$.

info

INTEGER.

If $info = 0$: successful exit

If $info > 0$: A problem occurred in $?larrv$. If $info = 5$, the Rayleigh Quotient Iteration failed to converge to full accuracy.

If $info < 0$: One of the called subroutines signaled an internal problem. Inspection of the corresponding parameter $info$ for further information is required.

- If $info = -1$, there is a problem in $?larrb$ when refining a child eigenvalue;
- If $info = -2$, there is a problem in $?larrf$ when computing the relatively robust representation (RRR) of a child. When a child is inside a tight cluster, it can be difficult to find an RRR. A partial remedy from the user's point of view is to make the parameter $minrgp$ smaller and recompile. However, as the orthogonality of the computed vectors is proportional to $1/minrgp$, you should be aware that you might be trading in precision when you decrease $minrgp$.
- If $info = -3$, there is a problem in $?larrb$ when refining a single eigenvalue after the Rayleigh correction was rejected.

See Also[?larrb](#)[?larre](#)[?larrf](#)**?lartg**

Generates a plane rotation with real cosine and real/complex sine.

Syntax

```
call slartg( f, g, cs, sn, r )
call dlartg( f, g, cs, sn, r )
call clartg( f, g, cs, sn, r )
call zlartg( f, g, cs, sn, r )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine generates a plane rotation so that

$$\begin{bmatrix} cs & sn \\ -\text{conj}(sn) & cs \end{bmatrix} \cdot \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $cs^2 + |sn|^2 = 1$

This is a slower, more accurate version of the BLAS Level 1 routine [?rotg](#), except for the following differences.

For slartg/dlartg:

f and g are unchanged on return;

If $g=0$, then $cs=1$ and $sn=0$;

If $f=0$ and $g \neq 0$, then $cs=0$ and $sn=1$ without doing any floating point operations (saves work in [?bdsqr](#) when there are zeros on the diagonal);

If f exceeds g in magnitude, cs will be positive.

For clartg/zlartg:

f and g are unchanged on return;

If $g=0$, then $cs=1$ and $sn=0$;

If $f=0$, then $cs=0$ and sn is chosen so that r is real.

Input Parameters

f, g	REAL for slartg DOUBLE PRECISION for dlartg COMPLEX for clartg DOUBLE COMPLEX for zlartg
The first and second component of vector to be rotated.	

Output Parameters

cs	REAL for slartg/clartg DOUBLE PRECISION for dlartg/zlartg
The cosine of the rotation.	
sn	REAL for slartg DOUBLE PRECISION for dlartg COMPLEX for clartg DOUBLE COMPLEX for zlartg
The sine of the rotation.	
r	REAL for slartg

DOUBLE PRECISION **for** `dlartg`
 COMPLEX **for** `clartg`
 DOUBLE COMPLEX **for** `zlartg`

The nonzero component of the rotated vector.

?lartgp

Generates a plane rotation so that the diagonal is nonnegative.

Syntax

FORTRAN 77:

```
call slartgp( f, g, cs, sn, r )
call dlartgp( f, g, cs, sn, r )
```

Fortran 95:

```
call lartgp( f, g, cs, sn, r )
```

C:

```
lapack_int LAPACKE_<?>lartgp ( <datatype> f, <datatype> g, <datatype> * cs, <datatype> * sn, <datatype> * r );
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine generates a plane rotation so that

$$\begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix} \cdot \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $cs^2 + sn^2 = 1$

This is a slower, more accurate version of the BLAS Level 1 routine `?rotg`, except for the following differences:

- f and g are unchanged on return.
- If $g=0$, then $cs=(+/-)1$ and $sn=0$.
- If $f=0$ and $g \neq 0$, then $cs=0$ and $sn=(+/-)1$.

The sign is chosen so that $r \geq 0$.

Input Parameters

The data types are given for the Fortran interface.

f, g	REAL for <code>slartgp</code>
	DOUBLE PRECISION for <code>dlartgp</code>

The first and second component of the vector to be rotated.

Output Parameters

<i>cs</i>	REAL for slartgp DOUBLE PRECISION for dlartgp
The cosine of the rotation.	
<i>sn</i>	REAL for slartgp DOUBLE PRECISION for dlartgp
The sine of the rotation.	
<i>r</i>	REAL for slartgp DOUBLE PRECISION for dlartgp
The nonzero component of the rotated vector.	

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine ?lartgp interface are as follows:

<i>f</i>	Holds the first component of the vector to be rotated.
<i>g</i>	Holds the second component of the vector to be rotated.
<i>cs</i>	Holds the cosine of the rotation.
<i>sn</i>	Holds the sine of the rotation.
<i>r</i>	Holds the nonzero component of the rotated vector.

See Also

[?rotg](#)
[?lartg](#)
[?lartgs](#)

?lartgs

Generates a plane rotation designed to introduce a bulge in implicit QR iteration for the bidiagonal SVD problem.

Syntax

FORTRAN 77:

```
call slartgs( x, y, sigma, cs, sn )
call dlartgs( x, y, sigma, cs, sn )
```

Fortran 95:

```
call lartgs( x,y,sigma,cs,sn )
```

C:

```
lapack_int LAPACKE_<?>lartgs (<datatype> x, <datatype> y, <datatype> sigma, <datatype>
* cs, <datatype> * sn);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine generates a plane rotation designed to introduce a bulge in Golub-Reinsch-style implicit QR iteration for the bidiagonal SVD problem. x and y are the top-row entries, and σ is the shift. The computed cs and sn define a plane rotation that satisfies the following:

$$\begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix} \cdot \begin{bmatrix} x^2 - \sigma \\ x * y \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

with r nonnegative.

If $x^2 - \sigma$ and $x * y$ are 0, the rotation is by $\pi/2$.

Input Parameters

The data types are given for the Fortran interface.

x, y	REAL for slartgs DOUBLE PRECISION for dlartgs
The (1,1) and (1,2) entries of an upper bidiagonal matrix, respectively.	
σ	REAL for slartgs DOUBLE PRECISION for dlartgs
	Shift

Output Parameters

cs	REAL for slartgs DOUBLE PRECISION for dlartgs
The cosine of the rotation.	
sn	REAL for slartgs DOUBLE PRECISION for dlartgs
	The sine of the rotation.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine ?lartgs interface are as follows:

<i>x</i>	Holds the (1,1) entry of an upper diagonal matrix.
<i>y</i>	Holds the (1,2) entry of an upper diagonal matrix.
<i>sigma</i>	Holds the shift.
<i>cs</i>	Holds the cosine of the rotation.
<i>sn</i>	Holds the sine of the rotation.

See Also

[?lartg](#)
[?lartgp](#)

?lartv

Applies a vector of plane rotations with real cosines and real/complex sines to the elements of a pair of vectors.

Syntax

```
call slartv( n, x, incx, y, incy, c, s, incc )
call dlartv( n, x, incx, y, incy, c, s, incc )
call clartv( n, x, incx, y, incy, c, s, incc )
call zlartv( n, x, incx, y, incy, c, s, incc )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine applies a vector of real/complex plane rotations with real cosines to elements of the real/complex vectors *x* and *y*. For $i = 1, 2, \dots, n$

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} := \begin{bmatrix} c(i) & s(i) \\ -\text{conj}(s(i)) & c(i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Input Parameters

<i>n</i>	INTEGER. The number of plane rotations to be applied.
<i>x</i> , <i>y</i>	REAL for slartv DOUBLE PRECISION for dlartv COMPLEX for clartv DOUBLE COMPLEX for zlartv
	Arrays, DIMENSION (1+(<i>n</i> -1)* <i>incx</i>) and (1+(<i>n</i> -1)* <i>incy</i>), respectively. The input vectors <i>x</i> and <i>y</i> .

<i>incx</i>	INTEGER. The increment between elements of <i>x</i> . <i>incx</i> > 0.
<i>incy</i>	INTEGER. The increment between elements of <i>y</i> . <i>incy</i> > 0.
<i>c</i>	REAL for slartv/clartv DOUBLE PRECISION for dlartv/zlartv Array, DIMENSION (1+(<i>n</i> -1)* <i>incc</i>). The cosines of the plane rotations.
<i>s</i>	REAL for slartv DOUBLE PRECISION for dlartv COMPLEX for clartv DOUBLE COMPLEX for zlartv Array, DIMENSION (1+(<i>n</i> -1)* <i>incc</i>). The sines of the plane rotations.
<i>incc</i>	INTEGER. The increment between elements of <i>c</i> and <i>s</i> . <i>incc</i> > 0.

Output Parameters

<i>x, y</i>	The rotated vectors <i>x</i> and <i>y</i> .
-------------	---

?laruv

Returns a vector of *n* random real numbers from a uniform distribution.

Syntax

```
call slaruv( iseed, n, x )
call dlaruv( iseed, n, x )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?laruv returns a vector of *n* random real numbers from a uniform (0,1) distribution (*n* ≤ 128).

This is an auxiliary routine called by ?larnv.

Input Parameters

<i>iseed</i>	INTEGER. Array, DIMENSION (4). On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and <i>iseed</i> (4) must be odd.
<i>n</i>	INTEGER. The number of random numbers to be generated. <i>n</i> ≤ 128.

Output Parameters

<i>x</i>	REAL for slaruv DOUBLE PRECISION for dlaruv Array, DIMENSION (<i>n</i>). The generated random numbers.
<i>seed</i>	On exit, the seed is updated.

?larz

Applies an elementary reflector (as returned by ?tzrzf) to a general matrix.

Syntax

```
call slarz( side, m, n, l, v, incv, tau, c, ldc, work )
call dlarz( side, m, n, l, v, incv, tau, c, ldc, work )
call clarz( side, m, n, l, v, incv, tau, c, ldc, work )
call zlarz( side, m, n, l, v, incv, tau, c, ldc, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?larz applies a real/complex elementary reflector H to a real/complex m -by- n matrix C , from either the left or the right. H is represented in the forms

$H = I - \tau u^* v^* v^T$ for real flavors and $H = I - \tau u^* v^* v^H$ for complex flavors,

where τ is a real/complex scalar and v is a real/complex vector, respectively.

If $\tau = 0$, then H is taken to be the unit matrix.

For complex flavors, to apply H^H (the conjugate transpose of H), supply `conjg(tau)` instead of τ .

H is a product of k elementary reflectors as returned by ?tzrzf.

Input Parameters

<i>side</i>	CHARACTER*1. If <i>side</i> = 'L': form $H*C$ If <i>side</i> = 'R': form $C*H$
<i>m</i>	INTEGER. The number of rows of the matrix C .
<i>n</i>	INTEGER. The number of columns of the matrix C .
<i>l</i>	INTEGER. The number of entries of the vector v containing the meaningful part of the Householder vectors. If <i>side</i> = 'L', $m \geq l \geq 0$, if <i>side</i> = 'R', $n \geq l \geq 0$.

v

REAL for slarz
DOUBLE PRECISION for dlarz
COMPLEX for clarz
DOUBLE COMPLEX for zlarz
Array, DIMENSION (1+(*l*-1)*abs(*incv*)).
The vector *v* in the representation of *H* as returned by ?tzrzf.
v is not used if *tau* = 0.

incv

INTEGER. The increment between elements of *v*.
incv ≠ 0.

tau

REAL for slarz
DOUBLE PRECISION for dlarz
COMPLEX for clarz
DOUBLE COMPLEX for zlarz
The value *tau* in the representation of *H*.

c

REAL for slarz
DOUBLE PRECISION for dlarz
COMPLEX for clarz
DOUBLE COMPLEX for zlarz
Array, DIMENSION (*ldc,n*).
On entry, the *m*-by-*n* matrix *C*.

ldc

INTEGER. The leading dimension of the array *c*.
ldc ≥ max(1, *m*).

work

REAL for slarz
DOUBLE PRECISION for dlarz
COMPLEX for clarz
DOUBLE COMPLEX for zlarz
Workspace array, DIMENSION
(*n*) if *side* = 'L' or
(*m*) if *side* = 'R'.

Output Parameters

c On exit, *C* is overwritten by the matrix *H*C* if *side* = 'L', or *C*H* if *side* = 'R'.

?larzb

Applies a block reflector or its transpose/conjugate-transpose to a general matrix.

Syntax

```
call slarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work,
ldwork )

call dlarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work,
ldwork )

call clarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work,
ldwork )

call zlarzb( side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, c, ldc, work,
ldwork )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine applies a real/complex block reflector H or its transpose H^T (or the conjugate transpose H^H for complex flavors) to a real/complex distributed m -by- n matrix C from the left or the right. Currently, only $storev = 'R'$ and $direct = 'B'$ are supported.

Input Parameters

<i>side</i>	CHARACTER*1. If <i>side</i> = 'L': apply H or H^T/H^H from the left If <i>side</i> = 'R': apply H or H^T/H^H from the right
<i>trans</i>	CHARACTER*1. If <i>trans</i> = 'N': apply H (No transpose) If <i>trans</i> ='C': apply H^H (conjugate transpose) If <i>trans</i> ='T': apply H^T (transpose transpose)
<i>direct</i>	CHARACTER*1. Indicates how H is formed from a product of elementary reflectors = 'F': $H = H(1)*H(2)*\dots*H(k)$ (forward, not supported) = 'B': $H = H(k)*\dots*H(2)*H(1)$ (backward)
<i>storev</i>	CHARACTER*1. Indicates how the vectors which define the elementary reflectors are stored: = 'C': Column-wise (not supported) = 'R': Row-wise.

m INTEGER. The number of rows of the matrix *C*.

n INTEGER. The number of columns of the matrix *C*.

k INTEGER. The order of the matrix *T* (equal to the number of elementary reflectors whose product defines the block reflector).

l INTEGER. The number of columns of the matrix *V* containing the meaningful part of the Householder reflectors.

If *side* = 'L', $m \geq l \geq 0$, if *side* = 'R', $n \geq l \geq 0$.

v REAL for slarzb
DOUBLE PRECISION for dlarzb
COMPLEX for clarzb
DOUBLE COMPLEX for zlarzb
Array, DIMENSION (*ldv*, *nv*).
If *storev* = 'C', *nv* = *k*;
if *storev* = 'R', *nv* = *l*.

ldv INTEGER. The leading dimension of the array *v*.
If *storev* = 'C', *ldv* $\geq l$; if *storev* = 'R', *ldv* $\geq k$.

t REAL for slarzb
DOUBLE PRECISION for dlarzb
COMPLEX for clarzb
DOUBLE COMPLEX for zlarzb
Array, DIMENSION (*ldt*, *k*). The triangular *k*-by-*k* matrix *T* in the representation of the block reflector.

ldt INTEGER. The leading dimension of the array *t*.
ldt $\geq k$.

c REAL for slarzb
DOUBLE PRECISION for dlarzb
COMPLEX for clarzb
DOUBLE COMPLEX for zlarzb
Array, DIMENSION (*ldc*, *n*). On entry, the *m*-by-*n* matrix *C*.

ldc INTEGER. The leading dimension of the array *c*.
ldc $\geq \max(1, m)$.

work REAL for slarzb
DOUBLE PRECISION for dlarzb
COMPLEX for clarzb
DOUBLE COMPLEX for zlarzb

Workspace array, DIMENSION (*ldwork*, *k*).

ldwork INTEGER. The leading dimension of the array *work*.
If *side* = 'L', *ldwork* $\geq \max(1, n)$;
if *side* = 'R', *ldwork* $\geq \max(1, m)$.

Output Parameters

c On exit, *C* is overwritten by H^*C , or H^T/H^H*C , or C^*H , or C^*H^T/H^H .

?larzt

Forms the triangular factor *T* of a block reflector $H = I - V*T^*V^H$

Syntax

```
call slarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call dlarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call clarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
call zlarzt( direct, storev, n, k, v, ldv, tau, t, ldt )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine forms the triangular factor *T* of a real/complex block reflector *H* of order $> n$, which is defined as a product of *k* elementary reflectors.

If *direct* = 'F', $H = H(1)*H(2)*\dots*H(k)$, and *T* is upper triangular.

If *direct* = 'B', $H = H(k)*\dots*H(2)*H(1)$, and *T* is lower triangular.

If *storev* = 'C', the vector which defines the elementary reflector $H(i)$ is stored in the *i*-th column of the array *v*, and $H = I - V*T^*V^T$ (for real flavors) or $H = I - V*T^*V^H$ (for complex flavors).

If *storev* = 'R', the vector which defines the elementary reflector $H(i)$ is stored in the *i*-th row of the array *v*, and $H = I - V^T*T^*V$ (for real flavors) or $H = I - V^{H^T}*T^*V$ (for complex flavors).

Currently, only *storev* = 'R' and *direct* = 'B' are supported.

Input Parameters

direct CHARACTER*1.

Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

If *direct* = 'F': $H = H(1)*H(2)*\dots*H(k)$ (forward, not supported)

If *direct* = 'B': $H = H(k)*\dots*H(2)*H(1)$ (backward)

storev CHARACTER*1.

Specifies how the vectors which define the elementary reflectors are stored (see also *Application Notes* below):

If $storev = 'C'$: column-wise (not supported)

If $storev = 'R'$: row-wise

n INTEGER. The order of the block reflector H . $n \geq 0$.

k INTEGER. The order of the triangular factor T (equal to the number of elementary reflectors). $k \geq 1$.

v REAL for `slarzt`

DOUBLE PRECISION for `dlarzt`

COMPLEX for `clarzt`

DOUBLE COMPLEX for `zlarzt`

Array, DIMENSION

(ldv, k) if $storev = 'C'$

(ldv, n) if $storev = 'R'$ The matrix V .

ldv INTEGER. The leading dimension of the array v .

If $storev = 'C'$, $ldv \geq \max(1, n)$;

if $storev = 'R'$, $ldv \geq k$.

τ REAL for `slarzt`

DOUBLE PRECISION for `dlarzt`

COMPLEX for `clarzt`

DOUBLE COMPLEX for `zlarzt`

Array, DIMENSION (k). $\tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$.

ldt INTEGER. The leading dimension of the output array t .

$ldt \geq k$.

Output Parameters

t REAL for `slarzt`

DOUBLE PRECISION for `dlarzt`

COMPLEX for `clarzt`

DOUBLE COMPLEX for `zlarzt`

Array, DIMENSION (ldt, k). The k -by- k triangular factor T of the block reflector. If $direct = 'F'$, T is upper triangular; if $direct = 'B'$, T is lower triangular. The rest of the array is not used.

v The matrix V . See *Application Notes* below.

Application Notes

The shape of the matrix V and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

direct = 'F' and storev = 'C': direct = 'F' and storev = 'R':

$$V = \begin{bmatrix} v_1 & v_2 & v_3 \\ \vdots & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ 1 & \ddots & \vdots \\ 1 & \ddots & \vdots \\ 1 & & \end{bmatrix}$$

 V
/ \

$$\begin{bmatrix} v_1 & v_2 & v_3 & v_1 & v_2 & v_3 & \cdots & \cdots & \cdots & 1 \\ v_1 & v_2 & v_3 & v_1 & v_2 & v_3 & \cdots & \cdots & \cdots & 1 \\ v_3 & v_2 & v_1 & v_3 & v_2 & v_1 & \cdots & \cdots & \cdots & 1 \end{bmatrix}$$

direct = 'B' and storev = 'C': direct = 'B' and storev = 'R':

$$V = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot \\ 1 & \cdot & \cdot \\ 1 & & \end{bmatrix}$$

 V
/ \

$$\begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & v_1 & v_2 & v_3 & v_4 & v_5 \\ \cdot & 1 & \cdot & \cdot & \cdot & v_1 & v_2 & v_3 & v_4 & v_5 \\ \cdot & \cdot & 1 & \cdot & \cdot & v_2 & v_3 & v_4 & v_5 & v_5 \end{bmatrix}$$

$$V = \begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix}$$

?las2

Computes singular values of a 2-by-2 triangular matrix.

Syntax

```
call slas2( f, g, h, ssmin, ssmax )
call dlas2( f, g, h, ssmin, ssmax )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?las2 computes the singular values of the 2-by-2 matrix

$$\begin{bmatrix} f & g \\ 0 & h \end{bmatrix}$$

On return, *ssmin* is the smaller singular value and *SSMAX* is the larger singular value.

Input Parameters

<i>f, g, h</i>	REAL for slas2 DOUBLE PRECISION for dlas2
	The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix, respectively.

Output Parameters

<i>ssmin, ssmax</i>	REAL for slas2 DOUBLE PRECISION for dlas2
	The smaller and the larger singular values, respectively.

Application Notes

Barring over/underflow, all output quantities are correct to within a few units in the last place (*ulps*), even in the absence of a guard digit in addition/subtraction. In ieee arithmetic, the code works correctly if one matrix element is infinite. Overflow will not occur unless the largest singular value itself overflows, or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.) Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?lascl

Multiples a general rectangular matrix by a real scalar defined as c_{to}/c_{from} .

Syntax

```
call slascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
```

```
call dlascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call clascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
call zlascl( type, kl, ku, cfrom, cto, m, n, a, lda, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lascl multiplies the m -by- n real/complex matrix A by the real scalar $c_{\text{to}}/c_{\text{from}}$. The operation is performed without over/underflow as long as the final result $c_{\text{to}} \cdot A(i, j) / c_{\text{from}}$ does not over/underflow.

type specifies that A may be full, upper triangular, lower triangular, upper Hessenberg, or banded.

Input Parameters

<i>type</i>	CHARACTER*1. This parameter specifies the storage type of the input matrix.
= 'G'	: A is a full matrix.
= 'L'	: A is a lower triangular matrix.
= 'U'	: A is an upper triangular matrix.
= 'H'	: A is an upper Hessenberg matrix.
= 'B'	: A is a symmetric band matrix with lower bandwidth kl and upper bandwidth ku and with the only the lower half stored
= 'Q'	: A is a symmetric band matrix with lower bandwidth kl and upper bandwidth ku and with the only the upper half stored.
= 'Z'	: A is a band matrix with lower bandwidth kl and upper bandwidth ku . See description of the ?gbtrf function for storage details.
<i>kl</i>	INTEGER. The lower bandwidth of A . Referenced only if <i>type</i> = 'B', 'Q' or 'Z'.
<i>ku</i>	INTEGER. The upper bandwidth of A . Referenced only if <i>type</i> = 'B', 'Q' or 'Z'.
<i>cfrom, cto</i>	REAL for slascl/clascl DOUBLE PRECISION for dlascl/zlascl The matrix A is multiplied by $cto/cfrom$. $A(i, j)$ is computed without over/underflow if the final result $cto \cdot A(i, j) / cfrom$ can be represented without over/underflow. <i>cfrom</i> must be nonzero.
<i>m</i>	INTEGER. The number of rows of the matrix A . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix A . $n \geq 0$.
<i>a</i>	REAL for slascl DOUBLE PRECISION for dlascl COMPLEX for clascl

DOUBLE COMPLEX for zlascl
 Array, DIMENSION (lda, n). The matrix to be multiplied by $cto/cfrom$. See type for the storage type.

lda INTEGER. The leading dimension of the array a .
 $lda \geq \max(1, m)$.

Output Parameters

a The multiplied matrix A .
 $info$ INTEGER.
 If $info = 0$ - successful exit
 If $info = -i < 0$, the i -th argument had an illegal value.

See Also

?gbtrf

?lasd0

Computes the singular values of a real upper bidiagonal n -by- m matrix B with diagonal d and off-diagonal e . Used by ?bdsdc.

Syntax

```
call slasd0(  $n, sqre, d, e, u, ldu, vt, ldvt, smlsiz, iwork, work, info$  )
call dlasd0(  $n, sqre, d, e, u, ldu, vt, ldvt, smlsiz, iwork, work, info$  )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

Using a divide and conquer approach, the routine ?lasd0 computes the singular value decomposition (SVD) of a real upper bidiagonal n -by- m matrix B with diagonal d and offdiagonal e , where $m = n + sqre$.

The algorithm computes orthogonal matrices U and VT such that $B = U^* S^* VT$. The singular values S are overwritten on d .

The related subroutine ?lasda computes only the singular values, and optionally, the singular vectors in compact form.

Input Parameters

n INTEGER. On entry, the row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array d .
 $sqre$ INTEGER. Specifies the column dimension of the bidiagonal matrix.
 If $sqre = 0$: the bidiagonal matrix has column dimension $m = n$.
 If $sqre = 1$: the bidiagonal matrix has column dimension $m = n+1$.

<i>d</i>	REAL for slasd0 DOUBLE PRECISION for dlasd0 Array, DIMENSION (<i>n</i>). On entry, <i>d</i> contains the main diagonal of the bidiagonal matrix.
<i>e</i>	REAL for slasd0 DOUBLE PRECISION for dlasd0 Array, DIMENSION (<i>m</i> -1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, <i>e</i> is destroyed.
<i>ldu</i>	INTEGER. On entry, leading dimension of the output array <i>u</i> .
<i>ldvt</i>	INTEGER. On entry, leading dimension of the output array <i>vt</i> .
<i>smlsiz</i>	INTEGER. On entry, maximum size of the subproblems at the bottom of the computation tree.
<i>iwork</i>	INTEGER. Workspace array, dimension must be at least (8 <i>n</i>).
<i>work</i>	REAL for slasd0 DOUBLE PRECISION for dlasd0 Workspace array, dimension must be at least (3 <i>m</i> ² +2 <i>m</i>).

Output Parameters

<i>d</i>	On exit <i>d</i> , If <i>info</i> = 0, contains singular values of the bidiagonal matrix.
<i>u</i>	REAL for slasd0 DOUBLE PRECISION for dlasd0 Array, DIMENSION at least (<i>ldq</i> , <i>n</i>). On exit, <i>u</i> contains the left singular vectors.
<i>vt</i>	REAL for slasd0 DOUBLE PRECISION for dlasd0 Array, DIMENSION at least (<i>ldvt</i> , <i>m</i>). On exit, <i>vt</i> ^T contains the right singular vectors.
<i>info</i>	INTEGER. If <i>info</i> = 0: successful exit. If <i>info</i> = - <i>i</i> < 0, the <i>i</i> -th argument had an illegal value. If <i>info</i> = 1, a singular value did not converge.

?lasd1

Computes the SVD of an upper bidiagonal matrix *B* of the specified size. Used by ?bdsdc.

Syntax

```
call slasd1( nl, nr, sqre, d, alpha, beta, u, ldu, vt, ldvt, idxq, iwork, work, info )
call dlasd1( nl, nr, sqre, d, alpha, beta, u, ldu, vt, ldvt, idxq, iwork, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the SVD of an upper bidiagonal n -by- m matrix B , where $n = nl + nr + 1$ and $m = n + sqre$.

The routine ?lasd1 is called from ?lasd0.

A related subroutine ?asd7 handles the case in which the singular values (and the singular vectors in factored form) are desired.

?lasd1 computes the SVD as follows:

$$VT = U(in)^* \begin{bmatrix} D1(in) & 0 & 0 & 0 \\ Z1^T & a & Z2^T & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix} * VT(in)$$

$$= U(out) * (D(out) \ 0) * VT(out)$$

where $Z^T = (Z1^T \ a \ Z2^T \ b) = u^T * VT^T$, and u is a vector of dimension m with *alpha* and *beta* in the $nl+1$ and $nl+2$ -th entries and zeros elsewhere; and the entry *b* is empty if *sqre* = 0.

The left singular vectors of the original matrix are stored in *u*, and the transpose of the right singular vectors are stored in *vt*, and the singular values are in *d*. The algorithm consists of three stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or when there are zeros in the *Z* vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine ?asd2.
2. The second stage consists of calculating the updated singular values. This is done by finding the square roots of the roots of the secular equation via the routine ?asd4 (as called by ?asd3). This routine also calculates the singular vectors of the current problem.
3. The final stage consists of computing the updated singular vectors directly using the updated singular values. The singular vectors for the current problem are multiplied with the singular vectors from the overall problem.

Input Parameters

nl INTEGER. The row dimension of the upper block.

$$nl \geq 1.$$

nr INTEGER. The row dimension of the lower block.

$$nr \geq 1.$$

sqre INTEGER.

If *sqre* = 0: the lower block is an *nr*-by-*nr* square matrix.

If $sqre = 1$: the lower block is an nr -by- $(nr+1)$ rectangular matrix. The bidiagonal matrix has row dimension $n = nl + nr + 1$, and column dimension $m = n + sqre$.

d

REAL for slasd1

DOUBLE PRECISION for dlasd1

Array, DIMENSION ($nl+nr+1$). $n = nl+nr+1$. On entry $d(1:nl, 1:nl)$ contains the singular values of the upper block; and $d(nl+2:n)$ contains the singular values of the lower block.

alpha

REAL for slasd1

DOUBLE PRECISION for dlasd1

Contains the diagonal element associated with the added row.

beta

REAL for slasd1

DOUBLE PRECISION for dlasd1

Contains the off-diagonal element associated with the added row.

u

REAL for slasd1

DOUBLE PRECISION for dlasd1

Array, DIMENSION (lDU, n). On entry $u(1:nl, 1:nl)$ contains the left singular vectors of the upper block; $u(nl+2:n, nl+2:n)$ contains the left singular vectors of the lower block.

*ldu*INTEGER. The leading dimension of the array U .

$lDU \geq \max(1, n)$.

vt

REAL for slasd1

DOUBLE PRECISION for dlasd1

Array, DIMENSION ($lDVT, m$), where $m = n + sqre$.

On entry $vt(1:nl+1, 1:nl+1)^T$ contains the right singular vectors of the upper block; $vt(nl+2:m, nl+2:m)^T$ contains the right singular vectors of the lower block.

*ldvt*INTEGER. The leading dimension of the array vt .

$lDVT \geq \max(1, M)$.

iwork

INTEGER.

Workspace array, DIMENSION ($4n$).

work

REAL for slasd1

DOUBLE PRECISION for dlasd1

Workspace array, DIMENSION ($3m_2 + 2m$).

Output Parameters

d

On exit $d(1:n)$ contains the singular values of the modified matrix.

<i>alpha</i>	On exit, the diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(D(I)))$, $I = 1, n$.
<i>beta</i>	On exit, the off-diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(D(I)))$, $I = 1, n$.
<i>u</i>	On exit <i>u</i> contains the left singular vectors of the bidiagonal matrix.
<i>vt</i>	On exit vt^T contains the right singular vectors of the bidiagonal matrix.
<i>idxq</i>	INTEGER Array, DIMENSION (<i>n</i>). Contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, $d(idxq(i = 1, n))$ will be in ascending order.
<i>info</i>	INTEGER. If <i>info</i> = 0: successful exit. If <i>info</i> = $-i < 0$, the <i>i</i> -th argument had an illegal value. If <i>info</i> = 1, a singular value did not converge.

?lasd2

Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.

Syntax

```
call slasd2( nl, nr, sqre, k, d, z, alpha, beta, u, ldu, vt, ldvt, dsigma, u2, ldu2,  

vt2, ldvt2, idxp, idxp, idxq, coltyp, info )  
  
call dlasd2( nl, nr, sqre, k, d, z, alpha, beta, u, ldu, vt, ldvt, dsigma, u2, ldu2,  

vt2, ldvt2, idxp, idxp, idxq, coltyp, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasd2 merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the Z vector. For each such occurrence the order of the related secular equation problem is reduced by one.

The routine ?lasd2 is called from ?lasd1.

Input Parameters

<i>nl</i>	INTEGER. The row dimension of the upper block. $nl \geq 1$.
<i>nr</i>	INTEGER. The row dimension of the lower block.

$nr \geq 1$.

sqre

INTEGER.

If $sqre = 0$): the lower block is an nr -by- nr square matrix

If $sqre = 1$): the lower block is an nr -by- $(nr+1)$ rectangular matrix. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.

d

REAL for slasd2

DOUBLE PRECISION for dlasd2

Array, DIMENSION (n). On entry *d* contains the singular values of the two submatrices to be combined.

alpha

REAL for slasd2

DOUBLE PRECISION for dlasd2

Contains the diagonal element associated with the added row.

beta

REAL for slasd2

DOUBLE PRECISION for dlasd2

Contains the off-diagonal element associated with the added row.

u

REAL for slasd2

DOUBLE PRECISION for dlasd2

Array, DIMENSION (lDU, n). On entry *u* contains the left singular vectors of two submatrices in the two square blocks with corners at $(1,1)$, (nl, nl) , and $(nl+2, nl+2)$, (n,n) .

ldu

INTEGER. The leading dimension of the array *u*.

$ldu \geq n$.

ldu2

INTEGER. The leading dimension of the output array *u2*. $ldu2 \geq n$.

vt

REAL for slasd2

DOUBLE PRECISION for dlasd2

Array, DIMENSION ($ldvt, m$). On entry, vt^T contains the right singular vectors of two submatrices in the two square blocks with corners at $(1,1)$, $(nl+1, nl+1)$, and $(nl+2, nl+2)$, (m, m) .

ldvt

INTEGER. The leading dimension of the array *vt*. $ldvt \geq m$.

ldvt2

INTEGER. The leading dimension of the output array *vt2*. $ldvt2 \geq m$.

idxp

INTEGER.

Workspace array, DIMENSION (n). This will contain the permutation used to place deflated values of *D* at the end of the array. On output *idxp*($2:k$) points to the nondeflated *d*-values and *idxp*($k+1:n$) points to the deflated singular values.

idx

INTEGER.

Workspace array, DIMENSION (n). This will contain the permutation used to sort the contents of d into ascending order.

coltyp

INTEGER.

Workspace array, DIMENSION (n). As workspace, this array contains a label that indicates which of the following types a column in the $u2$ matrix or a row in the $vt2$ matrix is:

- 1 : non-zero in the upper half only
- 2 : non-zero in the lower half only
- 3 : dense
- 4 : deflated.

idxq

INTEGER. Array, DIMENSION (n). This parameter contains the permutation that separately sorts the two sub-problems in D in the ascending order. Note that entries in the first half of this permutation must first be moved one position backwards and entries in the second half must have $nl+1$ added to their values.

Output Parameters

k

INTEGER. Contains the dimension of the non-deflated matrix, This is the order of the related secular equation. $1 \leq k \leq n$.

d

On exit D contains the trailing ($n-k$) updated singular values (those which were deflated) sorted into increasing order.

u

On exit u contains the trailing ($n-k$) updated left singular vectors (those which were deflated) in its last $n-k$ columns.

*z*REAL for `slasd2`DOUBLE PRECISION for `dlasd2`

Array, DIMENSION (n). On exit, z contains the updating row vector in the secular equation.

*dsigma*REAL for `slasd2`DOUBLE PRECISION for `dlassd2`

Array, DIMENSION (n). Contains a copy of the diagonal elements ($k-1$ singular values and one zero) in the secular equation.

*u2*REAL for `slasd2`DOUBLE PRECISION for `dlassd2`

Array, DIMENSION ($lDU2, n$). Contains a copy of the first $k-1$ left singular vectors which will be used by `?lasd3` in a matrix multiply (`?gemm`) to solve for the new left singular vectors. $u2$ is arranged into four blocks. The first block contains a column with 1 at $nl+1$ and zero everywhere else; the second block contains non-zero entries only at and above nl ; the third contains non-zero entries only below $nl+1$; and the fourth is dense.

<i>vt</i>	On exit, vt^T contains the trailing $(n-k)$ updated right singular vectors (those which were deflated) in its last $n-k$ columns. In case $sqre = 1$, the last row of <i>vt</i> spans the right null space.
<i>vt2</i>	<pre>REAL for slasd2 DOUBLE PRECISION for dlasd2</pre> <p>Array, DIMENSION ($ldvt2, n$). $vt2^T$ contains a copy of the first k right singular vectors which will be used by ?lasd3 in a matrix multiply (?gemm) to solve for the new right singular vectors. <i>vt2</i> is arranged into three blocks. The first block contains a row that corresponds to the special 0 diagonal element in <i>sigma</i>; the second block contains non-zeros only at and before $nl + 1$; the third block contains non-zeros only at and after $nl + 2$.</p>
<i>idxc</i>	INTEGER. Array, DIMENSION (n). This will contain the permutation used to arrange the columns of the deflated <i>u</i> matrix into three groups: the first group contains non-zero entries only at and above nl , the second contains non-zero entries only below $nl+2$, and the third is dense.
<i>coltyp</i>	On exit, it is an array of dimension 4, with $coltyp(i)$ being the dimension of the i -th type columns.
<i>info</i>	<pre>INTEGER. If info = 0): successful exit If info = -i < 0, the i-th argument had an illegal value.</pre>

?lasd3

Finds all square roots of the roots of the secular equation, as defined by the values in *D* and *Z*, and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.

Syntax

```
call slasd3( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu, u2, ldu2, vt, ldvt, vt2,
ldvt2, idxc, ctot, z, info )
call dlasd3( nl, nr, sqre, k, d, q, ldq, dsigma, u, ldu, u2, ldu2, vt, ldvt, vt2,
ldvt2, idxc, ctot, z, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasd3 finds all the square roots of the roots of the secular equation, as defined by the values in *D* and *Z*.

It makes the appropriate calls to ?lasd4 and then updates the singular vectors by matrix multiplication.

The routine ?lasd3 is called from ?lasd1.

Input Parameters

<i>nl</i>	INTEGER. The row dimension of the upper block. $nl \geq 1.$
<i>nr</i>	INTEGER. The row dimension of the lower block. $nr \geq 1.$
<i>sqrre</i>	INTEGER. If <i>sqrre</i> = 0): the lower block is an <i>nr</i> -by- <i>nr</i> square matrix. If <i>sqrre</i> = 1): the lower block is an <i>nr</i> -by-(<i>nr</i> +1) rectangular matrix. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqrre \geq n$ columns.
<i>k</i>	INTEGER. The size of the secular equation, $1 \leq k \leq n.$
<i>q</i>	REAL for slasd3 DOUBLE PRECISION for dlasd3 Workspace array , DIMENSION at least (<i>ldq</i> , <i>k</i>).
<i>ldq</i>	INTEGER. The leading dimension of the array <i>Q</i> . $ldq \geq k.$
<i>dsigma</i>	REAL for slasd3 DOUBLE PRECISION for dlasd3 Array , DIMENSION (<i>k</i>). The first <i>k</i> elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.
<i>ldu</i>	INTEGER. The leading dimension of the array <i>u</i> . $ldu \geq n.$
<i>u2</i>	REAL for slasd3 DOUBLE PRECISION for dlasd3 Array , DIMENSION (<i>ldu2</i> , <i>n</i>). The first <i>k</i> columns of this matrix contain the non-deflated left singular vectors for the split problem.
<i>ldu2</i>	INTEGER. The leading dimension of the array <i>u2</i> . $ldu2 \geq n.$
<i>ldvt</i>	INTEGER. The leading dimension of the array <i>vt</i> . $ldvt \geq n.$
<i>vt2</i>	REAL for slasd3 DOUBLE PRECISION for dlasd3 Array , DIMENSION (<i>ldvt2</i> , <i>n</i>).

The first k columns of $vt2'$ contain the non-deflated right singular vectors for the split problem.

ldvt2

INTEGER. The leading dimension of the array $vt2$.

$ldvt2 \geq n$.

idxc

INTEGER. Array, DIMENSION (n).

The permutation used to arrange the columns of u (and rows of vt) into three groups: the first group contains non-zero entries only at and above (or before) $n_1 + 1$; the second contains non-zero entries only at and below (or after) $n_1 + 2$; and the third is dense. The first column of u and the row of vt are treated separately, however. The rows of the singular vectors found by ?lasd4 must be likewise permuted before the matrix multiplies can take place.

ctot

INTEGER. Array, DIMENSION (4). A count of the total number of the various types of columns in u (or rows in vt), as described in *idxc*.

The fourth column type is any column which has been deflated.

z

REAL for slasd3

DOUBLE PRECISION for dlasd3

Array, DIMENSION (k). The first k elements of this array contain the components of the deflation-adjusted updating row vector.

Output Parameters

d

REAL for slasd3

DOUBLE PRECISION for dlasd3

Array, DIMENSION (k). On exit the square roots of the roots of the secular equation, in ascending order.

u

REAL for slasd3

DOUBLE PRECISION for dlasd3

Array, DIMENSION (lDU, n).

The last $n - k$ columns of this matrix contain the deflated left singular vectors.

vt

REAL for slasd3

DOUBLE PRECISION for dlasd3

Array, DIMENSION ($ldvt, m$).

The last $m - k$ columns of vt' contain the deflated right singular vectors.

vt2

Destroyed on exit.

z

Destroyed on exit.

info

INTEGER.

If $info = 0$): successful exit.

If $\text{info} = -i < 0$, the i -th argument had an illegal value.
 If $\text{info} = 1$, an singular value did not converge.

Application Notes

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

?lasd4

Computes the square root of the i -th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix. Used by ?bdsdc.

Syntax

```
call slasd4( n, i, d, z, delta, rho, sigma, work, info)
call dlasd4( n, i, d, z, delta, rho, sigma, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the square root of the i -th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix whose entries are given as the squares of the corresponding entries in the array d , and that $0 \leq d(i) < d(j)$ for $i < j$ and that $\rho > 0$. This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

$$\text{diag}(d) * \text{diag}(d) + \rho * Z * Z^T,$$

where the Euclidean norm of Z is equal to 1. The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

Input Parameters

n	INTEGER. The length of all arrays.
i	INTEGER. The index of the eigenvalue to be computed. $1 \leq i \leq n$.
d	REAL for slasd4 DOUBLE PRECISION for dlasd4 Array, DIMENSION (n). The original eigenvalues. They must be in order, $0 \leq d(i) < d(j)$ for $i < j$.
z	REAL for slasd4 DOUBLE PRECISION for dlasd4

Array, DIMENSION (n).

The components of the updating vector.

rho REAL for slasd4

DOUBLE PRECISION for dlasd4

The scalar in the symmetric updating formula.

work REAL for slasd4

DOUBLE PRECISION for dlasd4

Workspace array, DIMENSION (n).

If $n \neq 1$, *work* contains $(d(j) + \sigma_i)$ in its j -th component.

If $n = 1$, then *work(1)* = 1.

Output Parameters

delta REAL for slasd4

DOUBLE PRECISION for dlasd4

Array, DIMENSION (n).

If $n \neq 1$, *delta* contains $(d(j) - \sigma_i)$ in its j -th component.

If $n = 1$, then *delta (1)* = 1. The vector *delta* contains the information necessary to construct the (singular) eigenvectors.

sigma REAL for slasd4

DOUBLE PRECISION for dlasd4

The computed σ_i , the i -th updated eigenvalue.

info INTEGER.

= 0: successful exit

> 0: If *info* = 1, the updating process failed.

?lasd5

Computes the square root of the i -th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix. Used by ?bdsdc.

Syntax

```
call slasd5( i, d, z, delta, rho, dsigma, work )
call dlasd5( i, d, z, delta, rho, dsigma, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the square root of the i -th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix $\text{diag}(d) * \text{diag}(d) + rho * Z * Z^T$

The diagonal entries in the array d must satisfy $0 \leq d(i) < d(j)$ for $i < i$, rho must be greater than 0, and that the Euclidean norm of the vector Z is equal to 1.

Input Parameters

i	INTEGER. The index of the eigenvalue to be computed. $i = 1$ or $i = 2$.
d	REAL for slasd5 DOUBLE PRECISION for dlasd5 Array , dimension (2). The original eigenvalues, $0 \leq d(1) < d(2)$.
z	REAL for slasd5 DOUBLE PRECISION for dlasd5 Array , dimension (2). The components of the updating vector.
rho	REAL for slasd5 DOUBLE PRECISION for dlasd5 The scalar in the symmetric updating formula.
$work$	REAL for slasd5 DOUBLE PRECISION for dlasd5. Workspace array, dimension (2). Contains $(d(j) + sigma_i)$ in its j -th component.

Output Parameters

$delta$	REAL for slasd5 DOUBLE PRECISION for dlasd5. Array , dimension (2). Contains $(d(j) - sigma_i)$ in its j -th component. The vector $delta$ contains the information necessary to construct the eigenvectors.
$dsigma$	REAL for slasd5 DOUBLE PRECISION for dlasd5. The computed $sigma_i$, the i -th updated eigenvalue.

?lasd6

Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc.

Syntax

```
call slasd6( icompq, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr, givcol,  
ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork, info )
```

```
call dlasd6( icompq, nl, nr, sqre, d, vf, vl, alpha, beta, idxq, perm, givptr, givcol,  
ldgcol, givnum, ldgnum, poles, difl, difr, z, k, c, s, work, iwork, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasd6 computes the *SVD* of an updated upper bidiagonal matrix *B* obtained by merging two smaller ones by appending a row. This routine is used only for the problem which requires all singular values and optionally singular vector matrices in factored form. *B* is an *n*-by-*m* matrix with *n* = *nl* + *nr* + 1 and *m* = *n* + *sqre*. A related subroutine, ?lasd1, handles the case in which all singular values and singular vectors of the bidiagonal matrix are desired. ?lasd6 computes the *SVD* as follows:

$$B = U(in)^* \begin{bmatrix} D1(in) & 0 & 0 & 0 \\ z1' & a & z2' & b \\ 0 & 0 & D2(in) & 0 \end{bmatrix} * VT(in)$$

$$= U(out)^* (D(out))^* VT(out)$$

where $Z' = (Z1' \ a \ Z2' \ b) = u^* VT'$, and *u* is a vector of dimension *m* with *alpha* and *beta* in the *nl*+1 and *nl*+2-th entries and zeros elsewhere; and the entry *b* is empty if *sqre* = 0.

The singular values of *B* can be computed using *D1*, *D2*, the first components of all the right singular vectors of the lower block, and the last components of all the right singular vectors of the upper block. These components are stored and updated in *vf* and *vl*, respectively, in ?lasd6. Hence *U* and *VT* are not explicitly referenced.

The singular values are stored in *D*. The algorithm consists of two stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or if there is a zero in the *Z* vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine ?lasd7.
2. The second stage consists of calculating the updated singular values. This is done by finding the roots of the secular equation via the routine ?lasd4 (as called by ?lasd8). This routine also updates *vf* and *vl* and computes the distances between the updated singular values and the old singular values. ?lasd6 is called from ?lasda.

Input Parameters

<i>icompq</i>	INTEGER. Specifies whether singular vectors are to be computed in factored form: = 0: Compute singular values only = 1: Compute singular vectors in factored form as well.
<i>nl</i>	INTEGER. The row dimension of the upper block. $nl \geq 1$.

nr INTEGER. The row dimension of the lower block.
 $nr \geq 1$.

sqre INTEGER .
= 0: the lower block is an nr -by- nr square matrix.
= 1: the lower block is an nr -by- $(nr+1)$ rectangular matrix.
The bidiagonal matrix has row dimension $n = nl + nr + 1$, and column dimension $m = n + sqre$.

d REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, dimension ($nl + nr + 1$). On entry $d(1:nl, 1:nl)$ contains the singular values of the upper block, and $d(nl+2:n)$ contains the singular values of the lower block.

vf REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, dimension (m).
On entry, $vf(1:nl+1)$ contains the first components of all right singular vectors of the upper block; and $vf(nl+2:m)$ contains the first components of all right singular vectors of the lower block.

vl REAL for slasd6
DOUBLE PRECISION for dlasd6
Array, dimension (m).
On entry, $vl(1:nl+1)$ contains the last components of all right singular vectors of the upper block; and $vl(nl+2:m)$ contains the last components of all right singular vectors of the lower block.

alpha REAL for slasd6
DOUBLE PRECISION for dlasd6
Contains the diagonal element associated with the added row.

beta REAL for slasd6
DOUBLE PRECISION for dlasd6
Contains the off-diagonal element associated with the added row.

ldgcol INTEGER. The leading dimension of the output array *givcol*, must be at least n .

ldgnum INTEGER.
The leading dimension of the output arrays *givnum* and *poles*, must be at least n .

work REAL for slasd6
DOUBLE PRECISION for dlasd6

Workspace array, dimension ($4m$).

iwork INTEGER.

Workspace array, dimension ($3n$).

Output Parameters

d On exit $d(1:n)$ contains the singular values of the modified matrix.

vf On exit, *vf* contains the first components of all right singular vectors of the bidiagonal matrix.

vl On exit, *vl* contains the last components of all right singular vectors of the bidiagonal matrix.

alpha On exit, the diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(D(I)))$, $I = 1, n$.

beta On exit, the off-diagonal element associated with the added row deflated by $\max(\text{abs}(\text{alpha}), \text{abs}(\text{beta}), \text{abs}(D(I)))$, $I = 1, n$.

idxq INTEGER.

Array, dimension (n). This contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, $d(\text{idxq}(i = 1, n))$ will be in ascending order.

perm INTEGER.

Array, dimension (n). The permutations (from deflation and sorting) to be applied to each block. Not referenced if *icompq* = 0.

givptr INTEGER. The number of Givens rotations which took place in this subproblem. Not referenced if *icompq* = 0.

givcol INTEGER.

Array, dimension ($ldgcol, 2$). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if *icompq* = 0.

givnum REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension ($ldgnum, 2$). Each number indicates the C or S value to be used in the corresponding Givens rotation. Not referenced if *icompq* = 0.

poles REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension ($ldgnum, 2$). On exit, *poles*($1,*$) is an array containing the new singular values obtained from solving the secular equation, and *poles*($2,*$) is an array containing the poles in the secular equation. Not referenced if *icompq* = 0.

difl REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension (n). On exit, $difl(i)$ is the distance between i -th updated (undeflated) singular value and the i -th (undeflated) old singular value.

difr

REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension ($ldgnum, 2$) if $icompq = 1$ and dimension (n) if $icompq = 0$.

On exit, $difr(i, 1)$ is the distance between i -th updated (undeflated) singular value and the $i+1$ -th (undeflated) old singular value. If $icompq = 1$, $difr(1: k, 2)$ is an array containing the normalizing factors for the right singular vector matrix.

See ?lasd8 for details on *difl* and *difr*.

z

REAL for slasd6

DOUBLE PRECISION for dlasd6

Array, dimension (m).

The first elements of this array contain the components of the deflation-adjusted updating row vector.

k

INTEGER. Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. $1 \leq k \leq n$.

c

REAL for slasd6

DOUBLE PRECISION for dlasd6

c contains garbage if $sqre = 0$ and the C-value of a Givens rotation related to the right null space if $sqre = 1$.

s

REAL for slasd6

DOUBLE PRECISION for dlasd6

s contains garbage if $sqre = 0$ and the S-value of a Givens rotation related to the right null space if $sqre = 1$.

info

INTEGER.

= 0: successful exit.

< 0: if $info = -i$, the i -th argument had an illegal value.

> 0: if $info = 1$, an singular value did not converge

?lasd7

Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.

Syntax

```
call slasd7( icompq, nl, nr, sqre, k, d, z, zw, vf, vfw, vl, vlw, alpha, beta, dsigma,
idx, idxp, idxq, perm, givptr, givcol, ldgcol, givnum, ldgnum, c, s, info )
```

```
call dlasd7( icompq, nl, nr, sqre, k, d, z, zw, vf, vfw, vl, vlw, alpha, beta, dsigma,
idx, idxp, idxq, perm, givptr, givcol, ldgcol, givnum, ldgnum, c, s, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasd7 merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the Z vector. For each such occurrence the order of the related secular equation problem is reduced by one. ?lasd7 is called from ?lasd6.

Input Parameters

<i>icompq</i>	INTEGER. Specifies whether singular vectors are to be computed in compact form, as follows:
	= 0: Compute singular values only.
	= 1: Compute singular vectors of upper bidiagonal matrix in compact form.
<i>nl</i>	INTEGER. The row dimension of the upper block.
	<i>nl</i> \geq 1.
<i>nr</i>	INTEGER. The row dimension of the lower block.
	<i>nr</i> \geq 1.
<i>sqre</i>	INTEGER.
	= 0: the lower block is an <i>nr</i> -by- <i>nr</i> square matrix.
	= 1: the lower block is an <i>nr</i> -by-(<i>nr</i> +1) rectangular matrix. The bidiagonal matrix has <i>n</i> = <i>nl</i> + <i>nr</i> + 1 rows and <i>m</i> = <i>n</i> + <i>sqre</i> \geq <i>n</i> columns.
<i>d</i>	REAL for slasd7 DOUBLE PRECISION for dlasd7 Array, DIMENSION (<i>n</i>). On entry <i>d</i> contains the singular values of the two submatrices to be combined.
<i>zw</i>	REAL for slasd7 DOUBLE PRECISION for dlasd7 Array, DIMENSION (<i>m</i>). Workspace for <i>z</i> .
<i>vf</i>	REAL for slasd7 DOUBLE PRECISION for dlasd7 Array, DIMENSION (<i>m</i>). On entry, <i>vf</i> (1: <i>nl</i> +1) contains the first components of all right singular vectors of the upper block; and <i>vf</i> (<i>nl</i> +2: <i>m</i>) contains the first components of all right singular vectors of the lower block.

<i>vfw</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlassd7</code> Array, DIMENSION (<i>m</i>). Workspace for <i>vf</i> .
<i>vl</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlassd7</code> Array, DIMENSION (<i>m</i>). On entry, <i>vl</i> (1: <i>n</i> l+1) contains the last components of all right singular vectors of the upper block; and <i>vl</i> (<i>n</i> l+2: <i>m</i>) contains the last components of all right singular vectors of the lower block.
<i>VLW</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlassd7</code> Array, DIMENSION (<i>m</i>). Workspace for <i>VL</i> .
<i>alpha</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlassd7</code> . Contains the diagonal element associated with the added row.
<i>beta</i>	REAL for <code>slasd7</code> DOUBLE PRECISION for <code>dlassd7</code> Contains the off-diagonal element associated with the added row.
<i>idx</i>	INTEGER. Workspace array, DIMENSION (<i>n</i>). This will contain the permutation used to sort the contents of <i>d</i> into ascending order.
<i>idxp</i>	INTEGER. Workspace array, DIMENSION (<i>n</i>). This will contain the permutation used to place deflated values of <i>d</i> at the end of the array.
<i>idxq</i>	INTEGER. Array, DIMENSION (<i>n</i>). This contains the permutation which separately sorts the two sub-problems in <i>d</i> into ascending order. Note that entries in the first half of this permutation must first be moved one position backward; and entries in the second half must first have <i>n</i> l+1 added to their values.
<i>ldgcol</i>	INTEGER. The leading dimension of the output array <i>givcol</i> , must be at least <i>n</i> .
<i>ldgnum</i>	INTEGER. The leading dimension of the output array <i>givnum</i> , must be at least <i>n</i> .

Output Parameters

<i>k</i>	INTEGER. Contains the dimension of the non-deflated matrix, this is the order of the related secular equation. $1 \leq k \leq n$.
<i>d</i>	On exit, <i>d</i> contains the trailing ($n-k$) updated singular values (those which were deflated) sorted into increasing order.
<i>z</i>	REAL for slasd7 DOUBLE PRECISION for dlasd7. Array, DIMENSION (<i>m</i>). On exit, <i>Z</i> contains the updating row vector in the secular equation.
<i>vf</i>	On exit, <i>vf</i> contains the first components of all right singular vectors of the bidiagonal matrix.
<i>vl</i>	On exit, <i>vl</i> contains the last components of all right singular vectors of the bidiagonal matrix.
<i>dsigma</i>	REAL for slasd7 DOUBLE PRECISION for dlasd7. Array, DIMENSION (<i>n</i>). Contains a copy of the diagonal elements ($k-1$ singular values and one zero) in the secular equation.
<i>idxp</i>	On output, <i>idxp</i> ($2:k$) points to the nondeflated <i>d</i> -values and <i>idxp</i> ($k+1:n$) points to the deflated singular values.
<i>perm</i>	INTEGER. Array, DIMENSION (<i>n</i>). The permutations (from deflation and sorting) to be applied to each singular block. Not referenced if <i>icompq</i> = 0.
<i>givptr</i>	INTEGER. The number of Givens rotations which took place in this subproblem. Not referenced if <i>icompq</i> = 0.
<i>givcol</i>	INTEGER. Array, DIMENSION (<i>ldgcol</i> , 2). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if <i>icompq</i> = 0.
<i>givnum</i>	REAL for slasd7 DOUBLE PRECISION for dlasd7. Array, DIMENSION (<i>ldgnum</i> , 2). Each number indicates the C or S value to be used in the corresponding Givens rotation. Not referenced if <i>icompq</i> = 0.
<i>c</i>	REAL for slasd7. DOUBLE PRECISION for dlasd7.

If $sqre = 0$, then c contains garbage, and if $sqre = 1$, then c contains C -value of a Givens rotation related to the right null space.

s REAL for slasd7.

DOUBLE PRECISION for dlasd7.

If $sqre = 0$, then s contains garbage, and if $sqre = 1$, then s contains S -value of a Givens rotation related to the right null space.

$info$ INTEGER.

= 0: successful exit.

< 0: if $info = -i$, the i -th argument had an illegal value.

?lasd8

Finds the square roots of the roots of the secular equation, and stores, for each element in D , the distance to its two nearest poles. Used by ?bdsdc.

Syntax

```
call slasd8( icompq, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info )
call dlasd8( icompq, k, d, z, vf, vl, difl, difr, lddifr, dsigma, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasd8 finds the square roots of the roots of the secular equation, as defined by the values in $dsigma$ and z . It makes the appropriate calls to ?lasd4, and stores, for each element in d , the distance to its two nearest poles (elements in $dsigma$). It also updates the arrays vf and vl , the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd8 is called from ?lasd6.

Input Parameters

$icompq$

INTEGER. Specifies whether singular vectors are to be computed in factored form in the calling routine:

= 0: Compute singular values only.

= 1: Compute singular vectors in factored form as well.

k

INTEGER. The number of terms in the rational function to be solved by ?lasd4. $k \geq 1$.

z

REAL for slasd8

DOUBLE PRECISION for dlasd8.

Array, DIMENSION (k).

The first k elements of this array contain the components of the deflation-adjusted updating row vector.

vf REAL for slasd8
 DOUBLE PRECISION for dlasd8.
 Array, DIMENSION (*k*).
 On entry, *vf* contains information passed through dbede8.

vl REAL for slasd8
 DOUBLE PRECISION for dlasd8.
 Array, DIMENSION (*k*). On entry, *vl* contains information passed through dbede8.

lddifr INTEGER. The leading dimension of the output array *difr*, must be at least *k*.

dsigma REAL for slasd8
 DOUBLE PRECISION for dlasd8.
 Array, DIMENSION (*k*).
 The first *k* elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

work REAL for slasd8
 DOUBLE PRECISION for dlasd8.
 Workspace array, DIMENSION at least (3*k*).

Output Parameters

d REAL for slasd8
 DOUBLE PRECISION for dlasd8.
 Array, DIMENSION (*k*).
 On output, *D* contains the updated singular values.

z Updated on exit.

vf On exit, *vf* contains the first *k* components of the first components of all right singular vectors of the bidiagonal matrix.

vl On exit, *vl* contains the first *k* components of the last components of all right singular vectors of the bidiagonal matrix.

difl REAL for slasd8
 DOUBLE PRECISION for dlasd8.
 Array, DIMENSION (*k*). On exit, *difl(i) = d(i) - dsigma(i).*

difr REAL for slasd8
 DOUBLE PRECISION for dlasd8.
 Array,
 DIMENSION (*lddifr*, 2) if *icompq* = 1 and

DIMENSION (k) if $icompq = 0$.

On exit, $difr(i,1) = d(i) - dsigma(i+1)$, $difr(k,1)$ is not defined and will not be referenced. If $icompq = 1$, $difr(1:k,2)$ is an array containing the normalizing factors for the right singular vector matrix.

$dsigma$

The elements of this array may be very slightly altered in value.

$info$

INTEGER.

= 0: successful exit.

< 0: if $info = -i$, the i -th argument had an illegal value.

> 0: If $info = 1$, a singular value did not converge.

?lasd9

Finds the square roots of the roots of the secular equation, and stores, for each element in D , the distance to its two nearest poles. Used by ?bdsdc.

Syntax

```
call slasd9( icompq, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
call dlasd9( icompq, ldu, k, d, z, vf, vl, difl, difr, dsigma, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasd9 finds the square roots of the roots of the secular equation, as defined by the values in $dsigma$ and z . It makes the appropriate calls to ?lasd4, and stores, for each element in d , the distance to its two nearest poles (elements in $dsigma$). It also updates the arrays vf and vl , the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd9 is called from ?lasd7.

Input Parameters

$icompq$ INTEGER. Specifies whether singular vectors are to be computed in factored form in the calling routine:

If $icompq = 0$, compute singular values only;

If $icompq = 1$, compute singular vector matrices in factored form also.

k INTEGER. The number of terms in the rational function to be solved by slasd4. $k \geq 1$.

$dsigma$ REAL for slasd9

DOUBLE PRECISION for dlasd9.

Array, DIMENSION(k).

The first k elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

z REAL for slasd9
 DOUBLE PRECISION for dlasd9.
 Array, DIMENSION (k). The first k elements of this array contain the components of the deflation-adjusted updating row vector.

vf REAL for slasd9
 DOUBLE PRECISION for dlasd9.
 Array, DIMENSION(k). On entry, *vf* contains information passed through *sbede8*.

vl REAL for slasd9
 DOUBLE PRECISION for dlasd9.
 Array, DIMENSION(k). On entry, *vl* contains information passed through *sbede8*.

work REAL for slasd9
 DOUBLE PRECISION for dlasd9.
 Workspace array, DIMENSION at least (3k).

Output Parameters

d REAL for slasd9
 DOUBLE PRECISION for dlasd9.
 Array, DIMENSION(k). *d(i)* contains the updated singular values.

vf On exit, *vf* contains the first k components of the first components of all right singular vectors of the bidiagonal matrix.

vl On exit, *vl* contains the first k components of the last components of all right singular vectors of the bidiagonal matrix.

difl REAL for slasd9
 DOUBLE PRECISION for dlasd9.
 Array, DIMENSION (k).
 On exit, *difl(i) = d(i) - dsigma(i)*.

difr REAL for slasd9
 DOUBLE PRECISION for dlasd9.
 Array,
 DIMENSION (*ldu*, 2) if *icompq* = 1 and
 DIMENSION (k) if *icompq* = 0.
 On exit, *difr(i, 1) = d(i) - dsigma(i+1)*, *difr(k, 1)* is not defined and will not be referenced.
 If *icompq* = 1, *difr(1:k, 2)* is an array containing the normalizing factors for the right singular vector matrix.

info INTEGER.
 = 0: successful exit.
 < 0: if *info* = -*i*, the *i*-th argument had an illegal value.
 > 0: If *info* = 1, an singular value did not converge

?lasda

Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal *d* and off-diagonal *e*. Used by ?bdsdc.

Syntax

```
call slasda( icompq, smlsiz, n, sqre, d, e, u, ldu, vt, k, difl, difr, z, poles,  

givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )  

call dlasda( icompq, smlsiz, n, sqre, d, e, u, ldu, vt, k, difl, difr, z, poles,  

givptr, givcol, ldgcol, perm, givnum, c, s, work, iwork, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

Using a divide and conquer approach, ?lasda computes the singular value decomposition (SVD) of a real upper bidiagonal *n*-by-*m* matrix *B* with diagonal *d* and off-diagonal *e*, where *m* = *n* + *sqre*.

The algorithm computes the singular values in the SVD $B = U^* S^* V^T$. The orthogonal matrices *U* and *VT* are optionally computed in compact form. A related subroutine ?laasd0 computes the singular values and the singular vectors in explicit form.

Input Parameters

icompq INTEGER.
 Specifies whether singular vectors are to be computed in compact form, as follows:
 = 0: Compute singular values only.
 = 1: Compute singular vectors of upper bidiagonal matrix in compact form.

smlsiz INTEGER.
 The maximum size of the subproblems at the bottom of the computation tree.

n INTEGER. The row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array *d*.

sqre INTEGER. Specifies the column dimension of the bidiagonal matrix.
 If *sqre* = 0: the bidiagonal matrix has column dimension *m* = *n*
 If *sqre* = 1: the bidiagonal matrix has column dimension *m* = *n* + 1.

<i>d</i>	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (<i>n</i>). On entry, <i>d</i> contains the main diagonal of the bidiagonal matrix.
<i>e</i>	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (<i>m</i> - 1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, <i>e</i> is destroyed.
<i>ldu</i>	INTEGER. The leading dimension of arrays <i>u</i> , <i>vt</i> , <i>difl</i> , <i>difr</i> , <i>poles</i> , <i>givnum</i> , and <i>z</i> . <i>ldu</i> $\geq n$.
<i>ldgcol</i>	INTEGER. The leading dimension of arrays <i>givcol</i> and <i>perm</i> . <i>ldgcol</i> $\geq n$.
<i>work</i>	REAL for slasda DOUBLE PRECISION for dlasda. Workspace array, DIMENSION $(6n + (smlsiz+1)^2)$.
<i>iwork</i>	INTEGER. Workspace array, Dimension must be at least $(7n)$.

Output Parameters

<i>d</i>	On exit <i>d</i> , if <i>info</i> = 0, contains the singular values of the bidiagonal matrix.
<i>u</i>	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (<i>ldu</i> , <i>smlsiz</i>) if <i>icompq</i> = 1. Not referenced if <i>icompq</i> = 0. If <i>icompq</i> = 1, on exit, <i>u</i> contains the left singular vector matrices of all subproblems at the bottom level.
<i>vt</i>	REAL for slasda DOUBLE PRECISION for dlasda. Array, DIMENSION (<i>ldu</i> , <i>smlsiz</i> +1) if <i>icompq</i> = 1, and not referenced if <i>icompq</i> = 0. If <i>icompq</i> = 1, on exit, <i>vt</i> ' contains the right singular vector matrices of all subproblems at the bottom level.
<i>k</i>	INTEGER. Array, DIMENSION (<i>n</i>) if <i>icompq</i> = 1 and DIMENSION (1) if <i>icompq</i> = 0. If <i>icompq</i> = 1, on exit, <i>k</i> (<i>i</i>) is the dimension of the <i>i</i> -th secular equation on the computation tree.
<i>difl</i>	REAL for slasda

DOUBLE PRECISION for dlasda.

Array, DIMENSION (*lDU*, *nLVL*),

where $n\text{lvl} = \text{floor}(\log_2(n/\text{smlsiz}))$.

difr

REAL for slasda

DOUBLE PRECISION for dlasda.

Array,

DIMENSION (lDU , $2 \cdot nvl$) if $icompq = 1$ and

DIMENSION (n) if $icompq = 0$.

If $icompq = 1$, on exit, $difl(1:n, i)$ and $difr(1:n, 2i - 1)$ record distances between singular values on the i -th level and singular values on the $(i - 1)$ -th level, and $difr(1:n, 2i)$ contains the normalizing factors for the right singular vector matrix. See ?lasd8 for details.

z

REAL for slasda

DOUBLE PRECISION for dlasda.

Array,

DIMENSION (*ldu*, *nlvl*) if *icomprg* = 1 and

DIMENSION (n) if $icompq = 0$. The first k elements of $z(1, i)$ contain the components of the deflation-adjusted updating row vector for subproblems on the i -th level.

poles

REAL for slasda

DOUBLE PRECISION for dlasda

Array, DIMENSION (lDU , $2 * nLV1$)

if $icompq = 1$, and not referenced if $icompq = 0$. If $icompq = 1$, on exit, $\text{poles}(1, 2i - 1)$ and $\text{poles}(1, 2i)$ contain the new and old singular values involved in the secular equations on the i -th level.

givptr

INTEGER. Array, DIMENSION (n) if $icompq = 1$, and not referenced if $icompq = 0$. If $icompq = 1$, on exit, $givptr(i)$ records the number of Givens rotations performed on the i -th problem on the computation tree.

givcol

INTEGER.

Array, DIMENSION ($1dgcol$, $2*nlvl$) if $icompq = 1$, and not referenced if $icompq = 0$. If $icompq = 1$, on exit, for each i , $givcol(1, 2 i - 1)$ and $givcol(1, 2 i)$ record the locations of Givens rotations performed on the i -th level on the computation tree.

perm

INTEGER . Array, DIMENSION ($ldgcol$, $nlvl$) if $icompq = 1$, and not referenced if $icompq = 0$. If $icompq = 1$, on exit, $perm(1, i)$ records permutations done on the i -th level of the computation tree.

givnum

REAL for slasda

DOUBLE PRECISION for dlasda.

Array DIMENSION (*ldu*, $2*nvl$) if *icompq* = 1, and not referenced if *icompq* = 0. If *icompq* = 1, on exit, for each *i*, *givnum*(1, 2 *i* - 1) and *givnum*(1, 2 *i*) record the *C*- and *S*-values of Givens rotations performed on the *i*-th level on the computation tree.

c REAL for slasda

DOUBLE PRECISION for dlasda.

Array,

DIMENSION (*n*) if *icompq* = 1, and

DIMENSION (1) if *icompq* = 0.

If *icompq* = 1 and the *i*-th subproblem is not square, on exit, *c*(*i*) contains the *C*-value of a Givens rotation related to the right null space of the *i*-th subproblem.

s REAL for slasda

DOUBLE PRECISION for dlasda.

Array,

DIMENSION (*n*) *icompq* = 1, and

DIMENSION (1) if *icompq* = 0.

If *icompq* = 1 and the *i*-th subproblem is not square, on exit, *s*(*i*) contains the *S*-value of a Givens rotation related to the right null space of the *i*-th subproblem.

info INTEGER.

= 0: successful exit.

< 0: if *info* = -*i*, the *i*-th argument had an illegal value

> 0: If *info* = 1, an singular value did not converge

?lasdq

Computes the SVD of a real bidiagonal matrix with diagonal *d* and off-diagonal *e*. Used by ?bdsdc.

Syntax

```
call slasdq( uplo, sqre, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info )
```

```
call dlasdq( uplo, sqre, n, ncvt, nru, ncc, d, e, vt, ldvt, u, ldu, c, ldc, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine `?lasdq` computes the singular value decomposition (SVD) of a real (upper or lower) bidiagonal matrix with diagonal d and off-diagonal e , accumulating the transformations if desired. If B is the input bidiagonal matrix, the algorithm computes orthogonal matrices Q and P such that $B = Q^* S^* P^T$. The singular values S are overwritten on d .

The input matrix U is changed to $U^* Q$ if desired.

The input matrix VT is changed to $P^T * VT$ if desired.

The input matrix C is changed to $Q^T * C$ if desired.

Input Parameters

<code>uplo</code>	CHARACTER*1. On entry, <code>uplo</code> specifies whether the input bidiagonal matrix is upper or lower bidiagonal. If <code>uplo</code> = 'U' or 'u', B is upper bidiagonal; If <code>uplo</code> = 'L' or 'l', B is lower bidiagonal.
<code>sqre</code>	INTEGER. = 0: then the input matrix is n -by- n . = 1: then the input matrix is n -by- $(n+1)$ if <code>uplu</code> = 'U' and $(n+1)$ -by- n if <code>uplu</code> = 'L'. The bidiagonal matrix has $n = nl + nr + 1$ rows and $m = n + sqre \geq n$ columns.
<code>n</code>	INTEGER. On entry, <code>n</code> specifies the number of rows and columns in the matrix. <code>n</code> must be at least 0.
<code>ncvt</code>	INTEGER. On entry, <code>ncvt</code> specifies the number of columns of the matrix VT . <code>ncvt</code> must be at least 0.
<code>nru</code>	INTEGER. On entry, <code>nru</code> specifies the number of rows of the matrix U . <code>nru</code> must be at least 0.
<code>ncc</code>	INTEGER. On entry, <code>ncc</code> specifies the number of columns of the matrix C . <code>ncc</code> must be at least 0.
<code>d</code>	REAL for <code>slasdq</code> DOUBLE PRECISION for <code>dlassdq</code> . Array, DIMENSION (n). On entry, d contains the diagonal entries of the bidiagonal matrix.
<code>e</code>	REAL for <code>slasdq</code> DOUBLE PRECISION for <code>dlassdq</code> . Array, DIMENSION is $(n-1)$ if <code>sqre</code> = 0 and n if <code>sqre</code> = 1. On entry, the entries of e contain the off-diagonal entries of the bidiagonal matrix.
<code>vt</code>	REAL for <code>slasdq</code> DOUBLE PRECISION for <code>dlassdq</code> .

Array, DIMENSION (*ldvt, ncvt*). On entry, contains a matrix which on exit has been premultiplied by P^T , dimension n -by- $ncvt$ if *sqre* = 0 and $(n+1)$ -by- $ncvt$ if *sqre* = 1 (not referenced if *ncvt*=0).

ldvt INTEGER. On entry, *ldvt* specifies the leading dimension of *vt* as declared in the calling (sub) program. *ldvt* must be at least 1. If *ncvt* is nonzero, *ldvt* must also be at least *n*.

u REAL for *slasdq*

DOUBLE PRECISION for *dlassdq*.

Array, DIMENSION (*ldu, n*). On entry, contains a matrix which on exit has been postmultiplied by *Q*, dimension *nru*-by-*n* if *sqre* = 0 and *nru*-by- $(n+1)$ if *sqre* = 1 (not referenced if *nru*=0).

ldu INTEGER. On entry, *ldu* specifies the leading dimension of *u* as declared in the calling (sub) program. *ldu* must be at least $\max(1, nru)$.

c REAL for *slasdq*

DOUBLE PRECISION for *dlassdq*.

Array, DIMENSION (*ldc, ncc*). On entry, contains an *n*-by-*ncc* matrix which on exit has been premultiplied by *Q'*, dimension *n*-by-*ncc* if *sqre* = 0 and $(n+1)$ -by-*ncc* if *sqre* = 1 (not referenced if *ncc*=0).

ldc INTEGER. On entry, *ldc* specifies the leading dimension of *C* as declared in the calling (sub) program. *ldc* must be at least 1. If *ncc* is non-zero, *ldc* must also be at least *n*.

work REAL for *slasdq*

DOUBLE PRECISION for *dlassdq*.

Array, DIMENSION (4*n*). This is a workspace array. Only referenced if one of *ncvt*, *nru*, or *ncc* is nonzero, and if *n* is at least 2.

Output Parameters

d On normal exit, *d* contains the singular values in ascending order.

e On normal exit, *e* will contain 0. If the algorithm does not converge, *d* and *e* will contain the diagonal and superdiagonal entries of a bidiagonal matrix orthogonally equivalent to the one given as input.

vt On exit, the matrix has been premultiplied by *P'*.

u On exit, the matrix has been postmultiplied by *Q*.

c On exit, the matrix has been premultiplied by *Q'*.

info INTEGER. On exit, a value of 0 indicates a successful exit. If *info* < 0, argument number -*info* is illegal. If *info* > 0, the algorithm did not converge, and *info* specifies how many superdiagonals did not converge.

?lasdt

Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.

Syntax

```
call slasdt( n, lvl, nd, inode, ndiml, ndimr, msub )
call dlasdt( n, lvl, nd, inode, ndiml, ndimr, msub )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine creates a tree of subproblems for bidiagonal divide and conquer.

Input Parameters

<i>n</i>	INTEGER. On entry, the number of diagonal elements of the bidiagonal matrix.
<i>msub</i>	INTEGER. On entry, the maximum row dimension each subproblem at the bottom of the tree can be of.

Output Parameters

<i>lvl</i>	INTEGER. On exit, the number of levels on the computation tree.
<i>nd</i>	INTEGER. On exit, the number of nodes on the tree.
<i>inode</i>	INTEGER. Array, DIMENSION (<i>n</i>). On exit, centers of subproblems.
<i>ndiml</i>	INTEGER . Array, DIMENSION (<i>n</i>). On exit, row dimensions of left children.
<i>ndimr</i>	INTEGER . Array, DIMENSION (<i>n</i>). On exit, row dimensions of right children.

?laset

Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.

Syntax

```
call slaset( uplo, m, n, alpha, beta, a, lda )
call dlaset( uplo, m, n, alpha, beta, a, lda )
call claset( uplo, m, n, alpha, beta, a, lda )
call zlaset( uplo, m, n, alpha, beta, a, lda )
```

C:

```
lapack_int LAPACKE_<?>laset (int matrix_layout, char uplo, lapack_int m, lapack_int n,
<datatype> alpha, <datatype> beta, <datatype> * a, lapack_int lda);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine initializes an m -by- n matrix A to β on the diagonal and α on the off-diagonals.

Input Parameters

The data types are given for the Fortran interface.

<i>uplo</i>	CHARACTER*1. Specifies the part of the matrix A to be set. If <i>uplo</i> = 'U', upper triangular part is set; the strictly lower triangular part of A is not changed. If <i>uplo</i> = 'L': lower triangular part is set; the strictly upper triangular part of A is not changed. Otherwise: All of the matrix A is set.
<i>m</i>	INTEGER. The number of rows of the matrix A . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix A . $n \geq 0$.
<i>alpha, beta</i>	REAL for slaset DOUBLE PRECISION for dlaset COMPLEX for claset DOUBLE COMPLEX for zlaset. The constants to which the off-diagonal and diagonal elements are to be set, respectively.
<i>a</i>	REAL for slaset DOUBLE PRECISION for dlaset COMPLEX for claset DOUBLE COMPLEX for zlaset. Array, DIMENSION (<i>lda, n</i>). On entry, the m -by- n matrix A .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1,m)$.

Output Parameters

a On exit, the leading m -by- n submatrix of A is set as follows:

```

if uplo = 'U', A(i,j) = alpha, 1≤i≤j-1, 1≤j≤n,
if uplo = 'L', A(i,j) = alpha, j+1≤i≤m, 1≤j≤n,
otherwise, A(i,j) = alpha, 1≤i≤m, 1≤j≤n, i ≠ j,
and, for all uplo, A(i,i) = beta, 1≤i≤min(m, n).

```

?lasq1

Computes the singular values of a real square bidiagonal matrix. Used by ?bdsqr.

Syntax

```

call slasq1( n, d, e, work, info )
call dlasq1( n, d, e, work, info )

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasq1 computes the singular values of a real n -by- n bidiagonal matrix Z with diagonal d and off-diagonal e . The singular values are computed to high relative accuracy, in the absence of denormalization, underflow and overflow.

Input Parameters

<i>n</i>	INTEGER. The number of rows and columns in the matrix. $n \geq 0$.
<i>d</i>	REAL for slasq1 DOUBLE PRECISION for dlasq1. Array, DIMENSION (n). On entry, d contains the diagonal elements of the bidiagonal matrix whose SVD is desired.
<i>e</i>	REAL for slasq1 DOUBLE PRECISION for dlasq1. Array, DIMENSION (n). On entry, elements $e(1:n-1)$ contain the off-diagonal elements of the bidiagonal matrix whose SVD is desired.
<i>work</i>	REAL for slasq1 DOUBLE PRECISION for dlasq1. Workspace array, DIMENSION ($4n$).

Output Parameters

<i>d</i>	On normal exit, d contains the singular values in decreasing order.
----------	---

<i>e</i>	On exit, <i>e</i> is overwritten.
<i>info</i>	INTEGER. = 0: successful exit; < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value; > 0: the algorithm failed: = 1, a split was marked by a positive value in <i>e</i> ; = 2, current block of <i>Z</i> not diagonalized after $100n$ iterations (in inner while loop) - on exit the current contents of <i>d</i> and <i>e</i> represent a matrix with the same singular values as the matrix with which ?lasq1 was originally called, and which the calling subroutine could use to finish the computation, or even feed back into ?lasq1; = 3, termination criterion of outer while loop not met (program created more than <i>n</i> unreduced blocks).

?lasq2

Computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the quotient difference array z to high relative accuracy.
Used by ?bdsqr and ?steqr.

Syntax

```
call slasq2( n, z, info )
call dlasq2( n, z, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasq2 computes all the eigenvalues of the symmetric positive definite tridiagonal matrix associated with the quotient difference array *z* to high relative accuracy, in the absence of denormalization, underflow and overflow.

To see the relation of *z* to the tridiagonal matrix, let *L* be a unit lower bidiagonal matrix with subdiagonals *z*(2,4,6,...) and let *U* be an upper bidiagonal matrix with 1's above and diagonal *z*(1,3,5,...). The tridiagonal is *LU* or, if you prefer, the symmetric tridiagonal to which it is similar.

Input Parameters

<i>n</i>	INTEGER. The number of rows and columns in the matrix. $n \geq 0$.
<i>z</i>	REAL for slasq2 DOUBLE PRECISION for dlasq2. Array, DIMENSION (4 * <i>n</i>). On entry, <i>z</i> holds the quotient difference array.

Output Parameters

<i>z</i>	On exit, entries 1 to n hold the eigenvalues in decreasing order, $z(2n+1)$ holds the trace, and $z(2n+2)$ holds the sum of the eigenvalues. If $n > 2$, then $z(2n+3)$ holds the iteration count, $z(2n+4)$ holds $\text{ndivs}/\text{nint}^2$, and $z(2n+5)$ holds the percentage of shifts that failed.
<i>info</i>	<p>INTEGER.</p> <p>= 0: successful exit;</p> <p>< 0: if the i-th argument is a scalar and had an illegal value, then $info = -i$, if the i-th argument is an array and the j-entry had an illegal value, then $info = -(i*100 + j)$;</p> <p>> 0: the algorithm failed;</p> <p>= 1, a split was marked by a positive value in <i>e</i>;</p> <p>= 2, current block of <i>z</i> not diagonalized after $100*n$ iterations (in inner while loop) - On exit <i>z</i> holds a quotient difference array with the same eigenvalues as the <i>z</i> array on entry;</p> <p>= 3, termination criterion of outer while loop not met (program created more than n unreduced blocks).</p>

Application Notes

The routine `?lasq2` defines a logical variable, *ieee*, which is `.TRUE.` on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs, and `.FALSE.` otherwise. This variable is passed to `?lasq3`.

`?lasq3`

Checks for deflation, computes a shift and calls dqds.
Used by ?bdsqr.

Syntax

```
call slasq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee, ttype,
dmin1, dmin2, dn, dn1, dn2, g, tau )
call dlasq3( i0, n0, z, pp, dmin, sigma, desig, qmax, nfail, iter, ndiv, ieee, ttype,
dmin1, dmin2, dn, dn1, dn2, g, tau )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine `?lasq3` checks for deflation, computes a shift *tau*, and calls *dqds*. In case of failure, it changes shifts, and tries again until output is positive.

Input Parameters

<i>i0</i>	INTEGER. First index.
-----------	-----------------------

<i>n0</i>	INTEGER. Last index.
<i>z</i>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Array, DIMENSION ($4n$). <i>z</i> holds the <i>qd</i> array.
<i>pp</i>	INTEGER. <i>pp</i> =0 for ping, <i>pp</i> =1 for pong. <i>pp</i> =2 indicates that flipping was applied to the <i>Z</i> array and that the initial tests for deflation should not be performed.
<i>desig</i>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Lower order part of <i>sigma</i> .
<i>qmax</i>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Maximum value of <i>q</i> .
<i>ieee</i>	LOGICAL. Flag for <i>ieee</i> or non- <i>ieee</i> arithmetic (passed to <code>?lasq5</code>).
<i>ttype</i>	INTEGER. Shift type.
<i>dmin1, dmin2, dn, dn1, dn2, g</i>	REAL for <code>slasq3</code>
<i>tau</i>	DOUBLE PRECISION for <code>dlasq3</code> . These scalars are passed as arguments in order to save their values between calls to <code>?lasq3</code> .

Output Parameters

<i>dmin</i>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Minimum value of <i>d</i> .
<i>pp</i>	INTEGER. <i>pp</i> =0 for ping, <i>pp</i> =1 for pong. <i>pp</i> =2 indicates that flipping was applied to the <i>Z</i> array and that the initial tests for deflation should not be performed.
<i>sigma</i>	REAL for <code>slasq3</code> DOUBLE PRECISION for <code>dlasq3</code> . Sum of shifts used in the current segment.
<i>desig</i>	Lower order part of <i>sigma</i> .
<i>nfail</i>	INTEGER. Number of times shift was too big.
<i>iter</i>	INTEGER. Number of iterations.

<i>ndiv</i>	INTEGER. Number of divisions.
<i>ttype</i>	INTEGER. Shift type.
<i>dmin1, dmin2, dn, dn1, dn2, g</i>	REAL for <code>slasq3</code>
<i>tau</i>	DOUBLE PRECISION for <code>dlasq3</code> .
These scalars are passed as arguments in order to save their values between calls to <code>?lasq3</code> .	

?lasq4

Computes an approximation to the smallest eigenvalue using values of *d* from the previous transform. Used by `?bdsqr`.

Syntax

```
call slasq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau, ttype, g )
call dlasq4( i0, n0, z, pp, n0in, dmin, dmin1, dmin2, dn, dn1, dn2, tau, ttype, g )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine computes an approximation *tau* to the smallest eigenvalue using values of *d* from the previous transform.

Input Parameters

<i>i0</i>	INTEGER. First index.
<i>n0</i>	INTEGER. Last index.
<i>z</i>	REAL for <code>slasq4</code> DOUBLE PRECISION for <code>dlasq4</code> . Array, DIMENSION ($4n$). <i>z</i> holds the <i>qd</i> array.
<i>pp</i>	INTEGER. <i>pp=0</i> for ping, <i>pp=1</i> for pong.
<i>n0in</i>	INTEGER. The value of <i>n0</i> at start of <code>eigtest</code> .
<i>dmin</i>	REAL for <code>slasq4</code> DOUBLE PRECISION for <code>dlasq4</code> . Minimum value of <i>d</i> .
<i>dmin1</i>	REAL for <code>slasq4</code>

	DOUBLE PRECISION for dlasq4.
	Minimum value of d , excluding $d(n0)$.
$dmin2$	REAL for slasq4 DOUBLE PRECISION for dlasq4. Minimum value of d , excluding $d(n0)$ and $d(n0-1)$.
dn	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains $d(n)$.
$dn1$	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains $d(n-1)$.
$dn2$	REAL for slasq4 DOUBLE PRECISION for dlasq4. Contains $d(n-2)$.
g	REAL for slasq4 DOUBLE PRECISION for dlasq4. A scalar passed as an argument in order to save its value between calls to ?lasq4.

Output Parameters

τ	REAL for slasq4 DOUBLE PRECISION for dlasq4. Shift.
$ttype$	INTEGER. Shift type.
g	REAL for slasq4 DOUBLE PRECISION for dlasq4. A scalar passed as an argument in order to save its value between calls to ?lasq4.

?lasq5

Computes one dqds transform in ping-pong form.

Used by ?bdsqr and ?stegr.

Syntax

```
call slasq5( i0, n0, z, pp, tau, dmin, dmin1, dmin2, dn, dnm1, dnm2, ieee )
call dlasq5( i0, n0, z, pp, tau, dmin, dmin1, dmin2, dn, dnm1, dnm2, ieee )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes one dqds transform in ping-pong form: one version for ieee machines, another for non-ieee machines.

Input Parameters

<i>i0</i>	INTEGER. First index.
<i>n0</i>	INTEGER. Last index.
<i>z</i>	REAL for <code>slasq5</code> DOUBLE PRECISION for <code>dlasq5</code> . Array, DIMENSION ($4n$). <i>z</i> holds the <i>qd</i> array. <i>emin</i> is stored in <i>z</i> ($4*n0$) to avoid an extra argument.
<i>pp</i>	INTEGER. <i>pp</i> =0 for ping, <i>pp</i> =1 for pong.
<i>tau</i>	REAL for <code>slasq5</code> DOUBLE PRECISION for <code>dlasq5</code> . This is the shift.
<i>ieee</i>	LOGICAL. Flag for IEEE or non-IEEE arithmetic.

Output Parameters

<i>dmin</i>	REAL for <code>slasq5</code> DOUBLE PRECISION for <code>dlasq5</code> . Minimum value of <i>d</i> .
<i>dmin1</i>	REAL for <code>slasq5</code> DOUBLE PRECISION for <code>dlasq5</code> . Minimum value of <i>d</i> , excluding <i>d(n0)</i> .
<i>dmin2</i>	REAL for <code>slasq5</code> DOUBLE PRECISION for <code>dlasq5</code> . Minimum value of <i>d</i> , excluding <i>d(n0)</i> and <i>d(n0-1)</i> .
<i>dn</i>	REAL for <code>slasq5</code> DOUBLE PRECISION for <code>dlasq5</code> . Contains <i>d(n0)</i> , the last value of <i>d</i> .
<i>dnlm1</i>	REAL for <code>slasq5</code> DOUBLE PRECISION for <code>dlasq5</code> . Contains <i>d(n0-1)</i> .
<i>dnlm2</i>	REAL for <code>slasq5</code> DOUBLE PRECISION for <code>dlasq5</code> . Contains <i>d(n0-2)</i> .

?lasq6

Computes one *dqd* transform in ping-pong form. Used by ?bdsqr and ?stegr.

Syntax

```
call slasq6( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
call dlasq6( i0, n0, z, pp, dmin, dmin1, dmin2, dn, dnm1, dnm2 )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasq6 computes one *dqd* (shift equal to zero) transform in ping-pong form, with protection against underflow and overflow.

Input Parameters

i0 INTEGER. First index.

n0 INTEGER. Last index.

z REAL for slasq6
 DOUBLE PRECISION for dlasq6.
 Array, DIMENSION ($4n$). *Z* holds the *qd* array. *emin* is stored in *z*($4*n0$) to avoid an extra argument.

pp INTEGER. *pp*=0 for ping, *pp*=1 for pong.

Output Parameters

dmin REAL for slasq6
 DOUBLE PRECISION for dlasq6.
 Minimum value of *d*.

dmin1 REAL for slasq6
 DOUBLE PRECISION for dlasq6.
 Minimum value of *d*, excluding *d(n0)*.

dmin2 REAL for slasq6
 DOUBLE PRECISION for dlasq6.
 Minimum value of *d*, excluding *d(n0)* and *d(n0-1)*.

dn REAL for slasq6
 DOUBLE PRECISION for dlasq6. Contains *d(n0)*, the last value of *d*.

dnm1 REAL for slasq6

DOUBLE PRECISION for dlasq6. Contains $d(n0-1)$.

?lasr

Applies a sequence of plane rotations to a general rectangular matrix.

Syntax

```
call slasr( side, pivot, direct, m, n, c, s, a, lda )
call dlasr( side, pivot, direct, m, n, c, s, a, lda )
call clasr( side, pivot, direct, m, n, c, s, a, lda )
call zlasr( side, pivot, direct, m, n, c, s, a, lda )
```

Include Files

- Fortran: mkl.fi
 - C: mkl.h

Description

The routine applies a sequence of plane rotations to a real/complex matrix A , from the left or the right.

$A := P^* A$, when *side* = 'L' (Left-hand side)

A := *A***P'*, when *side* = 'R' (Right-hand side)

where P is an orthogonal matrix consisting of a sequence of plane rotations with $z = m$ when `side = 'L'` and $z = n$ when `side = 'R'`.

When `direct = 'F'` (Forward sequence), then

$$P = P(z-1) * \dots * P(2) * P(1),$$

and when `direct = 'B'` (Backward sequence), then

$$P = P(1) * P(2) * \dots * P(z-1),$$

where $P(k)$ is a plane rotation matrix defined by the 2-by-2 plane rotation:

... (1999) 111

When `pivot = 'v'` (Variable pivot), the rotation is performed for the plane $(k, k + 1)$, that is, $P(k)$ has the form

$$P(k) = \begin{bmatrix} 1 & & & & \\ & \dots & & & \\ & & 1 & & \\ & & & c(k) s(k) & \\ & & & -s(k) c(k) & \\ & & & & 1 \\ & & & & & \dots \\ & & & & & & 1 \end{bmatrix}$$

where $R(k)$ appears as a rank-2 modification to the identity matrix in rows and columns k and $k+1$.

When $pivot = 'T'$ (Top pivot), the rotation is performed for the plane $(1, k+1)$, so $P(k)$ has the form

$$P(k) = \begin{bmatrix} c(k) & & s(k) & & \\ & 1 & & & \\ & & \dots & & \\ & & & 1 & \\ & -s(k) & & & c(k) \\ & & & & 1 \\ & & & & & \dots \\ & & & & & & 1 \end{bmatrix}$$

where $R(k)$ appears in rows and columns k and $k+1$.

Similarly, when $pivot = 'B'$ (Bottom pivot), the rotation is performed for the plane (k, z) , giving $P(k)$ the form

$$P(k) = \begin{bmatrix} 1 & & & & & \\ & \dots & & & & \\ & & 1 & & & \\ & & & c(k) & & s(k) \\ & & & & 1 & \\ & & & & & \dots \\ & & & & & & 1 \\ & & & -s(k) & & & c(k) \end{bmatrix}$$

where $R(k)$ appears in rows and columns k and z . The rotations are performed without ever forming $P(k)$ explicitly.

Input Parameters

<i>side</i>	CHARACTER*1. Specifies whether the plane rotation matrix P is applied to A on the left or the right. = 'L': left, compute $A := P*A$ = 'R': right, compute $A:= A*P'$
<i>direct</i>	CHARACTER*1. Specifies whether P is a forward or backward sequence of plane rotations. = 'F': forward, $P = P(z-1)*...*P(2)*P(1)$ = 'B': backward, $P = P(1)*P(2)*...*P(z-1)$
<i>pivot</i>	CHARACTER*1. Specifies the plane for which $P(k)$ is a plane rotation matrix. = 'V': Variable pivot, the plane $(k, k+1)$ = 'T': Top pivot, the plane $(1, k+1)$ = 'B': Bottom pivot, the plane (k, z)
<i>m</i>	INTEGER. The number of rows of the matrix A . If $m \leq 1$, an immediate return is effected.
<i>n</i>	INTEGER. The number of columns of the matrix A . If $n \leq 1$, an immediate return is effected.
<i>c, s</i>	REAL for slasr/clasr DOUBLE PRECISION for dlasr/zlasr. Arrays, DIMENSION ($m-1$) if <i>side</i> = 'L', ($n-1$) if <i>side</i> = 'R'. <i>c(k)</i> and <i>s(k)</i> contain the cosine and sine of the plane rotations respectively that define the 2-by-2 plane rotation part ($R(k)$) of the $P(k)$ matrix as described above in <i>Description</i> .
<i>a</i>	REAL for slasr DOUBLE PRECISION for dlasr COMPLEX for clasr DOUBLE COMPLEX for zlasr. Array, DIMENSION (<i>lda, n</i>). The m -by- n matrix A .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, m)$.

Output Parameters

a On exit, *A* is overwritten by P^*A if *side* = 'R', or by A^*P if *side* = 'L'.

?lasrt

Sorts numbers in increasing or decreasing order.

Syntax

```
call slasrt( id, n, d, info )
```

```
call dlasrt( id, n, d, info )
```

C:

```
lapack_int LAPACKE_<?>lasrt (char id, lapack_int n, <datatype> * d);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasrt sorts the numbers in *d* in increasing order (if *id* = 'I') or in decreasing order (if *id* = 'D'). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20 . Dimension of stack limits *n* to about 2^{32} .

Input Parameters

The data types are given for the Fortran interface.

<i>id</i>	CHARACTER*1. = 'I': sort <i>d</i> in increasing order; = 'D': sort <i>d</i> in decreasing order.
-----------	--

<i>n</i>	INTEGER. The length of the array <i>d</i> .
----------	---

<i>d</i>	REAL for slasrt DOUBLE PRECISION for dlasrt. On entry, the array to be sorted.
----------	--

Output Parameters

<i>d</i>	On exit, <i>d</i> has been sorted into increasing order ($d(1) \leq \dots \leq d(n)$) or into decreasing order ($d(1) \geq \dots \geq d(n)$), depending on <i>id</i> .
----------	--

<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value.
-------------	--

?lassq

Updates a sum of squares represented in scaled form.

Syntax

```
call slassq( n, x, incx, scale, sumsq )
call dlassq( n, x, incx, scale, sumsq )
call classq( n, x, incx, scale, sumsq )
call zlassq( n, x, incx, scale, sumsq )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The real routines slassq/dlassq return the values *scl* and *smsq* such that

$$scl^2 * smsq = x(1)^2 + \dots + x(n)^2 + scale^2 * sumsq,$$

where $x(i) = x(1 + (i - 1) * incx)$.

The value of *sumsq* is assumed to be non-negative and *scl* returns the value

$$scl = \max(scale, \text{abs}(x(i))).$$

Values *scale* and *sumsq* must be supplied in *scale* and *sumsq*, and *scl* and *smsq* are overwritten on *scale* and *sumsq*, respectively.

The complex routines classq/zlassq return the values *scl* and *ssq* such that

$$scl^2 * ssq = x(1)^2 + \dots + x(n)^2 + scale^2 * sumsq,$$

where $x(i) = \text{abs}(x(1 + (i - 1) * incx))$.

The value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy $1.0 \leq ssq \leq sumsq + 2n$

scale is assumed to be non-negative and *scl* returns the value

$$scl = \max(scale, \text{abs}(\text{real}(x(i))), \text{abs}(\text{aimag}(x(i)))).$$

Values *scale* and *sumsq* must be supplied in *scale* and *sumsq*, and *scl* and *ssq* are overwritten on *scale* and *sumsq*, respectively.

All routines ?lassq make only one pass through the vector *x*.

Input Parameters

n INTEGER. The number of elements to be used from the vector *x*.

x REAL for slassq

DOUBLE PRECISION for dlassq

COMPLEX for classq

DOUBLE COMPLEX for zlassq.

The vector for which a scaled sum of squares is computed: $x(i) = x(1 + (i-1) * incx), 1 \leq i \leq n$.

<i>incx</i>	INTEGER. The increment between successive values of the vector <i>x</i> . <i>incx</i> > 0.
<i>scale</i>	REAL for <i>slassq/classq</i> DOUBLE PRECISION for <i>dlassq/zlassq</i> . On entry, the value <i>scale</i> in the equation above.
<i>sumsq</i>	REAL for <i>slassq/classq</i> DOUBLE PRECISION for <i>dlassq/zlassq</i> . On entry, the value <i>sumsq</i> in the equation above.

Output Parameters

<i>scale</i>	On exit, <i>scale</i> is overwritten with <i>sc</i> , the scaling factor for the sum of squares.
<i>sumsq</i>	For real flavors: On exit, <i>sumsq</i> is overwritten with the value <i>smsq</i> in the equation above. For complex flavors: On exit, <i>sumsq</i> is overwritten with the value <i>ssq</i> in the equation above.

?lasv2

Computes the singular value decomposition of a 2-by-2 triangular matrix.

Syntax

```
call slasv2( f, g, h, ssmin, ssmax, snr, csr, snl, csl )
call dlasv2( f, g, h, ssmin, ssmax, snr, csr, snl, csl )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasv2 computes the singular value decomposition of a 2-by-2 triangular matrix

$$\begin{bmatrix} f & g \\ 0 & h \end{bmatrix}$$

On return, $\text{abs}(\text{ssmax})$ is the larger singular value, $\text{abs}(\text{ssmin})$ is the smaller singular value, and (csl, snl) and (csr, snr) are the left and right singular vectors for $\text{abs}(\text{ssmax})$, giving the decomposition

$$\begin{bmatrix} \text{csl} & \text{snl} \\ -\text{snl} & \text{csl} \end{bmatrix} \begin{bmatrix} f & g \\ 0 & h \end{bmatrix} \begin{bmatrix} \text{csr} & -\text{snr} \\ \text{snr} & \text{csr} \end{bmatrix} = \begin{bmatrix} \text{ssmax} & 0 \\ 0 & \text{ssmin} \end{bmatrix}$$

Input Parameters

f, g, h

REAL for `slasv2`
DOUBLE PRECISION for `dlasv2`.
The (1,1), (1,2) and (2,2) elements of the 2-by-2 matrix, respectively.

Output Parameters

ssmin, ssmax

REAL for `slasv2`
DOUBLE PRECISION for `dlasv2`.
 $\text{abs}(ssmin)$ and $\text{abs}(ssmax)$ is the smaller and the larger singular value, respectively.

snl, csl

REAL for `slasv2`
DOUBLE PRECISION for `dlasv2`.
The vector (*csl, snl*) is a unit left singular vector for the singular value $\text{abs}(ssmax)$.

snr, csr

REAL for `slasv2`
DOUBLE PRECISION for `dlasv2`.
The vector (*csr, snr*) is a unit right singular vector for the singular value $\text{abs}(ssmax)$.

Application Notes

Any input parameter may be aliased with any output parameter.

Barring over/underflow and assuming a guard digit in subtraction, all output quantities are correct to within a few units in the last place (ulp).

In ieee arithmetic, the code works correctly if one matrix element is infinite. Overflow will not occur unless the largest singular value itself overflows or is within a few ulps of overflow. (On machines with partial overflow, like the Cray, overflow may occur if the largest singular value is within a factor of 2 of overflow.) Underflow is harmless if underflow is gradual. Otherwise, results may correspond to a matrix modified by perturbations of size near the underflow threshold.

?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```
call slaswp( n, a, lda, k1, k2, ipiv, incx )
call dlaswp( n, a, lda, k1, k2, ipiv, incx )
call claswp( n, a, lda, k1, k2, ipiv, incx )
call zlaswp( n, a, lda, k1, k2, ipiv, incx )
```

C:

```
lapack_int LAPACKE_<?>laswp (int matrix_layout, lapack_int n, <datatype> * a,
lapack_int lda, lapack_int k1, lapack_int k2, const lapack_int * ipiv, lapack_int
incx);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine performs a series of row interchanges on the matrix A . One row interchange is initiated for each of rows $k1$ through $k2$ of A .

Input Parameters

The data types are given for the Fortran interface.

<i>n</i>	INTEGER. The number of columns of the matrix A .
<i>a</i>	REAL for slaswp DOUBLE PRECISION for dlaswp COMPLEX for claswp DOUBLE COMPLEX for zlaswp. Array, DIMENSION (<i>lda, n</i>). On entry, the matrix of column dimension <i>n</i> to which the row interchanges will be applied.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> .
<i>k1</i>	INTEGER. The first element of <i>ipiv</i> for which a row interchange will be done.
<i>k2</i>	INTEGER. The last element of <i>ipiv</i> for which a row interchange will be done.
<i>ipiv</i>	INTEGER. Array, DIMENSION ($k2 \times incx $). The vector of pivot indices. Only the elements in positions $k1$ through $k2$ of <i>ipiv</i> are accessed. $ipiv(k) = l$ implies rows k and l are to be interchanged.
<i>incx</i>	INTEGER. The increment between successive values of <i>ipiv</i> . If <i>ipiv</i> is negative, the pivots are applied in reverse order.

Output Parameters

<i>a</i>	On exit, the permuted matrix.
----------	-------------------------------

?lasy2

Solves the Sylvester matrix equation where the matrices are of order 1 or 2.

Syntax

```
call slasy2( ltranl, ltranr, isgn, n1, n2, tl, ldltl, tr, ldtr, b, ldb, scale, x, ldx,
xnorm, info )
```

```
call dlasly2( ltranl, ltranr, isgn, n1, n2, tl, ldltl, tr, ldtr, b, ldb, scale, x, ldx,
xnorm, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine solves for the $n1$ -by- $n2$ matrix X , $1 \leq n1, n2 \leq 2$, in

$$\text{op}(TL) * X + isgn * X * \text{op}(TR) = scale * B,$$

where

TL is $n1$ -by- $n1$,

TR is $n2$ -by- $n2$,

B is $n1$ -by- $n2$,

and $isgn = 1$ or -1 . $\text{op}(T) = T$ or T^T , where T^T denotes the transpose of T .

Input Parameters

ltranl

LOGICAL.

On entry, *ltranl* specifies the $\text{op}(TL)$:

- = .FALSE., $\text{op}(TL) = TL$,
- = .TRUE., $\text{op}(TL) = (TL)^T$.

ltranr

LOGICAL.

On entry, *ltranr* specifies the $\text{op}(TR)$:

- = .FALSE., $\text{op}(TR) = TR$,
- = .TRUE., $\text{op}(TR) = (TR)^T$.

isgn

INTEGER. On entry, *isgn* specifies the sign of the equation as described before. *isgn* may only be 1 or -1.

n1

INTEGER. On entry, *n1* specifies the order of matrix TL .

n1 may only be 0, 1 or 2.

n2

INTEGER. On entry, *n2* specifies the order of matrix TR .

n2 may only be 0, 1 or 2.

tl

REAL for slasy2

DOUBLE PRECISION for dlasly2.

Array, DIMENSION (*ldtl*,2).

On entry, *tl* contains an $n1$ -by- $n1$ matrix TL .

<i>ldt1</i>	INTEGER. The leading dimension of the matrix <i>TL</i> . $ldt1 \geq \max(1, n1).$
<i>tr</i>	REAL for <code>slasy2</code> DOUBLE PRECISION for <code>dlassy2</code> . Array, DIMENSION (<i>ldtr</i> ,2). On entry, <i>tr</i> contains an n_2 -by- n_2 matrix <i>TR</i> .
<i>ldtr</i>	INTEGER. The leading dimension of the matrix <i>TR</i> . $ldtr \geq \max(1, n2).$
<i>b</i>	REAL for <code>slasy2</code> DOUBLE PRECISION for <code>dlassy2</code> . Array, DIMENSION (<i>ldb</i> ,2). On entry, the n_1 -by- n_2 matrix <i>B</i> contains the right-hand side of the equation.
<i>ldb</i>	INTEGER. The leading dimension of the matrix <i>B</i> . $ldb \geq \max(1, n1).$
<i>idx</i>	INTEGER. The leading dimension of the output matrix <i>X</i> . $idx \geq \max(1, n1).$

Output Parameters

<i>scale</i>	REAL for <code>slasy2</code> DOUBLE PRECISION for <code>dlassy2</code> . On exit, <i>scale</i> contains the scale factor. <i>scale</i> is chosen less than or equal to 1 to prevent the solution overflowing.
<i>x</i>	REAL for <code>slasy2</code> DOUBLE PRECISION for <code>dlassy2</code> . Array, DIMENSION (<i>idx</i> ,2). On exit, <i>x</i> contains the n_1 -by- n_2 solution.
<i>xnorm</i>	REAL for <code>slasy2</code> DOUBLE PRECISION for <code>dlassy2</code> . On exit, <i>xnorm</i> is the infinity-norm of the solution.
<i>info</i>	INTEGER. On exit, <i>info</i> is set to 0: successful exit. 1: <i>TL</i> and <i>TR</i> have too close eigenvalues, so <i>TL</i> or <i>TR</i> is perturbed to get a nonsingular equation.

NOTE

For higher speed, this routine does not check the inputs for errors.

?lasyf

Computes a partial factorization of a symmetric matrix, using the diagonal pivoting method.

Syntax

```
call slasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call dlasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call clasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlasyf( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lasyf computes a partial factorization of a symmetric matrix A using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

If $uplo = 'U'$:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}' & U_{22}' \end{bmatrix} \text{ if } uplo = 'U', \text{ or}$$

$uplo = 'L'$

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}' & L_{21}' \\ 0 & I \end{bmatrix} \text{ if } uplo = 'L'$$

where the order of D is at most nb .

The actual order is returned in the argument kb , and is either nb or $nb-1$, or n if $n \leq nb$.

This is an auxiliary routine called by ?sytrf. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if $uplo = 'U'$) or A_{22} (if $uplo = 'L'$).

Input Parameters

$uplo$

CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

- = 'U': Upper triangular
- = 'L': Lower triangular

n

INTEGER. The order of the matrix A . $n \geq 0$.

nb

INTEGER. The maximum number of columns of the matrix A that should be factored. nb should be at least 2 to allow for 2-by-2 pivot blocks.

a REAL for `slasyf`
 DOUBLE PRECISION for `dlassyf`
 COMPLEX for `clasyf`
 DOUBLE COMPLEX for `zlassyf`.
 Array, DIMENSION (lda, n). If $uplo = 'U'$, the leading n -by- n upper triangular part of *a* contains the upper triangular part of the matrix A , and the strictly lower triangular part of *a* is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of *a* contains the lower triangular part of the matrix A , and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

w REAL for `slasyf`
 DOUBLE PRECISION for `dlassyf`
 COMPLEX for `clasyf`
 DOUBLE COMPLEX for `zlassyf`.
 Workspace array, DIMENSION (ldw, nb).

ldw INTEGER. The leading dimension of the array *w*. $ldw \geq \max(1, n)$.

Output Parameters

kb INTEGER. The number of columns of A that were actually factored *kb* is either $nb-1$ or nb , or n if $n \leq nb$.

a On exit, *a* contains details of the partial factorization.

ipiv INTEGER. Array, DIMENSION (n). Details of the interchanges and the block structure of D .

If $uplo = 'U'$, only the last *kb* elements of *ipiv* are set;

if $uplo = 'L'$, only the first *kb* elements are set.

If $ipiv(k) > 0$, then rows and columns k and $ipiv(k)$ were interchanged and $D(k, k)$ is a 1-by-1 diagonal block.

If $uplo = 'U'$ and $ipiv(k) = ipiv(k-1) < 0$, then rows and columns $k-1$ and $-ipiv(k)$ were interchanged and $D(k-1:k, k-1:k)$ is a 2-by-2 diagonal block.

If $uplo = 'L'$ and $ipiv(k) = ipiv(k+1) < 0$, then rows and columns $k+1$ and $-ipiv(k)$ were interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.

info

INTEGER.

= 0: successful exit

> 0: if $info = k$, $D(k, k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular.

?lasyf_rook

Computes a partial factorization of a complex symmetric matrix, using the bounded Bunch-Kaufman diagonal pivoting method.

Syntax

```
call slasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call dlasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call clasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlasyf_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine `?lasyf_rook` computes a partial factorization of a complex symmetric matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method. The partial factorization has the form:

$$\begin{aligned} A &= \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}' & U_{22}' \end{bmatrix} \text{ if } \text{uplo} = 'U', \text{ or} \\ A &= \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}' & L_{21}' \\ 0 & I \end{bmatrix} \text{ if } \text{uplo} = 'L' \end{aligned}$$

where the order of D is at most nb .

The actual order is returned in the argument kb , and is either nb or $nb-1$, or n if $n \leq nb$.

This is an auxiliary routine called by `?sytrf_rook`. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if $\text{uplo} = 'U'$) or A_{22} (if $\text{uplo} = 'L'$).

Input Parameters

<code>uplo</code>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is stored: = 'U': Upper triangular = 'L': Lower triangular
<code>n</code>	INTEGER. The order of the matrix A . $n \geq 0$.
<code>nb</code>	INTEGER. The maximum number of columns of the matrix A that should be factored. nb should be at least 2 to allow for 2-by-2 pivot blocks.
<code>a</code>	REAL for <code>slasyf_rook</code>

DOUBLE PRECISION for `dlasyf_rook`
 COMPLEX for `clasyf_rook`
 DOUBLE COMPLEX for `zlassyf_rook`.

Array, DIMENSION (lda, n). If $uplo = 'U'$, the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

lda INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

w REAL for `slasyf_rook`
 DOUBLE PRECISION for `dlasyf_rook`
 COMPLEX for `clasyf_rook`
 DOUBLE COMPLEX for `zlassyf_rook`.
Workspace array, DIMENSION (ldw, nb).

ldw INTEGER. The leading dimension of the array w . $ldw \geq \max(1, n)$.

Output Parameters

kb INTEGER. The number of columns of A that were actually factored kb is either $nb-1$ or nb , or n if $n \leq nb$.

a On exit, a contains details of the partial factorization.

$ipiv$ INTEGER. Array, DIMENSION (n). Details of the interchanges and the block structure of D .

If $uplo = 'U'$, only the last kb elements of $ipiv$ are set;

if $uplo = 'L'$, only the first kb elements are set.

If $ipiv(k) > 0$, then rows and columns k and $ipiv(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $uplo = 'U'$ and $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, then rows and columns k and $-ipiv(k)$ were interchanged, rows and columns $k - 1$ and $-ipiv(k - 1)$ were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.

If $uplo = 'L'$ and $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns k and $-ipiv(k)$ were interchanged, rows and columns $k + 1$ and $-ipiv(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

$info$ INTEGER.

= 0: successful exit

> 0: if $info = k$, $D(k, k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular.

?lahef

Computes a partial factorization of a complex Hermitian indefinite matrix, using the diagonal pivoting method.

Syntax

```
call clahef( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlahef( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lahef computes a partial factorization of a complex Hermitian matrix A , using the Bunch-Kaufman diagonal pivoting method. The partial factorization has the form:

If $uplo = 'U'$:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}^H & U_{22}^H \end{bmatrix} \text{ if } uplo = 'U', \text{ or}$$

If $uplo = 'U'$:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}' & L_{21}' \\ 0 & I \end{bmatrix} \text{ if } uplo = 'L'$$

where the order of D is at most nb .

The actual order is returned in the argument kb , and is either nb or $nb-1$, or n if $n \leq nb$.

Note that U^H denotes the conjugate transpose of U .

This is an auxiliary routine called by ?hetrf. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if $uplo = 'U'$) or A_{22} (if $uplo = 'L'$).

Input Parameters

$uplo$

CHARACTER*1.

Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

- = 'U': upper triangular
- = 'L': lower triangular

n

INTEGER. The order of the matrix A . $n \geq 0$.

nb

INTEGER. The maximum number of columns of the matrix A that should be factored. nb should be at least 2 to allow for 2-by-2 pivot blocks.

a COMPLEX for `clahef`
 DOUBLE COMPLEX for `zlahef`.
 Array, DIMENSION (*lda, n*).
 On entry, the Hermitian matrix *A*.
 If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.
 If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

lda INTEGER. The leading dimension of the array *a*. *lda* $\geq \max(1, n)$.

w COMPLEX for `clahef`
 DOUBLE COMPLEX for `zlahef`.
 Workspace array, DIMENSION (*ldw, nb*).

ldw INTEGER. The leading dimension of the array *w*. *ldw* $\geq \max(1, n)$.

Output Parameters

kb INTEGER. The number of columns of *A* that were actually factored *kb* is either *nb*-1 or *nb*, or *n* if *n* $\leq nb$.

a On exit, *A* contains details of the partial factorization.

ipiv INTEGER.
 Array, DIMENSION (*n*). Details of the interchanges and the block structure of *D*.
 If *uplo* = 'U', only the last *kb* elements of *ipiv* are set;
 if *uplo* = 'L', only the first *kb* elements are set.
 If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) are interchanged and *D*(*k, k*) is a 1-by-1 diagonal block.
 If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k*-1) < 0, then rows and columns *k*-1 and -*ipiv*(*k*) are interchanged and *D*(*k*-1:k, *k*-1:k) is a 2-by-2 diagonal block.
 If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k*+1) < 0, then rows and columns *k*+1 and -*ipiv*(*k*) are interchanged and *D*(*k*:*k*+1, *k*:*k*+1) is a 2-by-2 diagonal block.

info INTEGER.
 = 0: successful exit
 > 0: if *info* = *k*, *D*(*k, k*) is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular.

?lahef_rook

Computes a partial factorization of a complex Hermitian indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method.

Syntax

```
call clahef_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
call zlahef_rook( uplo, n, nb, kb, a, lda, ipiv, w, ldw, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lahef_rook computes a partial factorization of a complex Hermitian matrix A , using the bounded Bunch-Kaufman ("rook") diagonal pivoting method. The partial factorization has the form:

If $uplo = 'U'$:

$$A = \begin{bmatrix} I & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & D \end{bmatrix} \begin{bmatrix} I & 0 \\ U_{12}^H & U_{22}^H \end{bmatrix} \text{ if } uplo = 'U', \text{ or}$$

If $uplo = 'L'$:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}' & L_{21}' \\ 0 & I \end{bmatrix} \text{ if } uplo = 'L'$$

where the order of D is at most nb .

The actual order is returned in the argument kb , and is either nb or $nb-1$, or n if $n \leq nb$.

Note that U^H denotes the conjugate transpose of U .

This is an auxiliary routine called by ?hetrf_rook. It uses blocked code (calling Level 3 BLAS) to update the submatrix A_{11} (if $uplo = 'U'$) or A_{22} (if $uplo = 'L'$).

Input Parameters

$uplo$	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: = ' U ': upper triangular = ' L ': lower triangular
n	INTEGER. The order of the matrix A . $n \geq 0$.
nb	INTEGER. The maximum number of columns of the matrix A that should be factored. nb should be at least 2 to allow for 2-by-2 pivot blocks.

a COMPLEX for `clahef_rook`
 DOUBLE COMPLEX for `zlahef_rook`.
 Array, DIMENSION (*lda*, *n*).
 On entry, the Hermitian matrix *A*.
 If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *A* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *A* is not referenced.
 If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *A* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *A* is not referenced.

lda INTEGER. The leading dimension of the array *a*. *lda* $\geq \max(1, n)$.

w COMPLEX for `clahef_rook`
 DOUBLE COMPLEX for `zlahef_rook`.
 Workspace array, DIMENSION (*ldw*, *nb*).

ldw INTEGER. The leading dimension of the array *w*. *ldw* $\geq \max(1, n)$.

Output Parameters

kb INTEGER. The number of columns of *A* that were actually factored *kb* is either *nb*-1 or *nb*, or *n* if *n* $\leq nb$.

a On exit, *A* contains details of the partial factorization.

ipiv INTEGER.
 Array, DIMENSION (*n*). Details of the interchanges and the block structure of *D*.
 If *uplo* = 'U', only the last *kb* elements of *ipiv* are set;
 if *uplo* = 'L', only the first *kb* elements are set.
 If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) are interchanged and *D*(*k*, *k*) is a 1-by-1 diagonal block.
 If *uplo* = 'U' and *ipiv*(*k*) < 0 and *ipiv*(*k*-1) < 0, then rows and columns *k* and -*ipiv*(*k*) are interchanged, rows and columns *k*-1 and -*ipiv*(*k*-1) are interchanged, and *D*_{*k*-1:*k*, *k*-1:*k*} is a 2-by-2 diagonal block.
 If *uplo* = 'L' and *ipiv*(*k*) < 0 and *ipiv*(*k*+1) < 0, then rows and columns *k* and -*ipiv*(*k*) are interchanged, rows and columns *k*+1 and -*ipiv*(*k*+1) are interchanged, and *D*_{*k*:*k*+1, *k*:*k*+1} is a 2-by-2 diagonal block.

info INTEGER.
 = 0: successful exit
 > 0: if *info* = *k*, *D*(*k*, *k*) is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular.

?latbs

Solves a triangular banded system of equations.

Syntax

```
call slatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
call dlatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
call clatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
call zlatbs( uplo, trans, diag, normin, n, kd, ab, ldab, x, scale, cnorm, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine solves one of the triangular systems

$$A^*x = s^*b, \text{ or } A^T*x = s^*b, \text{ or } A^H*x = s^*b \text{ (for complex flavors)}$$

with scaling to prevent overflow, where A is an upper or lower triangular band matrix. Here A^T denotes the transpose of A , A^H denotes the conjugate transpose of A , x and b are n -element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine ?tbsv is called. If the matrix A is singular ($A(j, j)=0$ for some j), then s is set to 0 and a non-trivial solution to $A^*x = 0$ is returned.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular. = 'U': upper triangular = 'L': lower triangular
<i>trans</i>	CHARACTER*1. Specifies the operation applied to A . = 'N': solve $A^*x = s^*b$ (no transpose) = 'T': solve $A^T*x = s^*b$ (transpose) = 'C': solve $A^H*x = s^*b$ (conjugate transpose)
<i>diag</i>	CHARACTER*1. Specifies whether the matrix A is unit triangular = 'N': non-unit triangular = 'U': unit triangular
<i>normin</i>	CHARACTER*1. Specifies whether $cnorm$ is set. = 'Y': $cnorm$ contains the column norms on entry;

= 'N': *cnorm* is not set on entry. On exit, the norms are computed and stored in *cnorm*.

n INTEGER. The order of the matrix *A*. $n \geq 0$.

kd INTEGER. The number of subdiagonals or superdiagonals in the triangular matrix *A*. $kb \geq 0$.

ab REAL for *slatbs*

DOUBLE PRECISION for *dlatbs*

COMPLEX for *clatbs*

DOUBLE COMPLEX for *zlatbs*.

Array, DIMENSION (*ldab*, *n*).

The upper or lower triangular band matrix *A*, stored in the first $kb+1$ rows of the array. The *j*-th column of *A* is stored in the *j*-th column of the array *ab* as follows:

```
if uplo = 'U', ab(kd+1+i -j, j) = A(i, j) for  $\max(1, j-kd) \leq i \leq j$ ;  
if uplo = 'L', ab(1+i -j, j) = A(i, j) for  $j \leq i \leq \min(n, j+kd)$ .
```

ldab INTEGER. The leading dimension of the array *ab*. $ldab \geq kb+1$.

x REAL for *slatbs*

DOUBLE PRECISION for *dlatbs*

COMPLEX for *clatbs*

DOUBLE COMPLEX for *zlatbs*.

Array, DIMENSION (*n*).

On entry, the right hand side *b* of the triangular system.

cnorm REAL for *slatbs/clatbs*

DOUBLE PRECISION for *dlatbs/zlatbs*.

Array, DIMENSION (*n*).

If *NORMIN* = 'Y', *cnorm* is an input argument and *cnorm(j)* contains the norm of the off-diagonal part of the *j*-th column of *A*.

If *trans* = 'N', *cnorm(j)* must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm(j)* must be greater than or equal to the 1-norm.

Output Parameters

scale REAL for *slatbs/clatbs*

DOUBLE PRECISION for *dlatbs/zlatbs*.

The scaling factor *s* for the triangular system as described above. If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to *Ax* = 0.

<i>cnorm</i>	If <i>normin</i> = 'N', <i>cnorm</i> is an output argument and <i>cnorm(j)</i> returns the 1-norm of the off-diagonal part of the <i>j</i> -th column of <i>A</i> .
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = - <i>k</i> , the <i>k</i> -th argument had an illegal value

?latdf

Uses the LU factorization of the *n*-by-*n* matrix computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate.

Syntax

```
call slatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpivot )
call dlatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpivot )
call clatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpivot )
call zlatdf( ijob, n, z, ldz, rhs, rdsum, rdscal, ipiv, jpivot )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?latdf uses the LU factorization of the *n*-by-*n* matrix *Z* computed by ?getc2 and computes a contribution to the reciprocal Dif-estimate by solving $Z^*x = b$ for *x*, and choosing the right-hand side *b* such that the norm of *x* is as large as possible. On entry *rhs* = *b* holds the contribution from earlier solved subsystems, and on return *rhs* = *x*.

The factorization of *Z* returned by ?getc2 has the form $Z = P^*L^*U^*Q$, where *P* and *Q* are permutation matrices. *L* is lower triangular with unit diagonal elements and *U* is upper triangular.

Input Parameters

<i>ijob</i>	INTEGER. <i>ijob</i> = 2: First compute an approximative null-vector <i>e</i> of <i>Z</i> using ?gecon, <i>e</i> is normalized, and solve for $Z^*x = \pm e - f$ with the sign giving the greater value of 2-norm(<i>x</i>). This option is about 5 times as expensive as default. <i>ijob</i> ≠ 2 (default): Local look ahead strategy where all entries of the right-hand side <i>b</i> is chosen as either +1 or -1 .
<i>n</i>	INTEGER. The number of columns of the matrix <i>Z</i> .
<i>z</i>	REAL for slatdf/clatdf DOUBLE PRECISION for dlatdf/zlatdf. Array, DIMENSION (<i>ldz</i> , <i>n</i>)

On entry, the *LU* part of the factorization of the n -by- n matrix Z computed by ?getc2: $Z = P^* L^* U^* Q$.

ldz INTEGER. The leading dimension of the array Z . $lda \geq \max(1, n)$.

rhs REAL for slatdf/clatdf

DOUBLE PRECISION for dlatdf/zlatdf.

Array, DIMENSION (n).

On entry, *rhs* contains contributions from other subsystems.

rdsum REAL for slatdf/clatdf

DOUBLE PRECISION for dlatdf/zlatdf.

On entry, the sum of squares of computed contributions to the *Dif*-estimate under computation by ?tgsvL, where the scaling factor *rdscal* has been factored out. If *trans* = 'T', *rdsum* is not touched.

Note that *rdsum* only makes sense when ?tgsv2 is called by ?tgsvL.

rdscal REAL for slatdf/clatdf

DOUBLE PRECISION for dlatdf/zlatdf.

On entry, scaling factor used to prevent overflow in *rdsum*.

If *trans* = 'T', *rdscal* is not touched.

Note that *rdscal* only makes sense when ?tgsv2 is called by ?tgsvL.

ipiv INTEGER.

Array, DIMENSION (n).

The pivot indices; for $1 \leq i \leq n$, row i of the matrix has been interchanged with row *ipiv*(i).

jpivot INTEGER.

Array, DIMENSION (n).

The pivot indices; for $1 \leq j \leq n$, column j of the matrix has been interchanged with column *jpivot*(j).

Output Parameters

rhs

On exit, *rhs* contains the solution of the subsystem with entries according to the value of *ijob*.

rdsum

On exit, the corresponding sum of squares updated with the contributions from the current sub-system.

If *trans* = 'T', *rdsum* is not touched.

rdscal

On exit, *rdscal* is updated with respect to the current contributions in *rdsum*.

If *trans* = 'T', *rdscal* is not touched.

?latps

Solves a triangular system of equations with the matrix held in packed storage.

Syntax

```
call slatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call dlatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call clatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
call zlatps( uplo, trans, diag, normin, n, ap, x, scale, cnorm, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?latps solves one of the triangular systems

$A^*x = s^*b$, or $A^T*x = s^*b$, or $A^H*x = s^*b$ (for complex flavors)

with scaling to prevent overflow, where A is an upper or lower triangular matrix stored in packed form. Here A^T denotes the transpose of A , A^H denotes the conjugate transpose of A , x and b are n -element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem does not cause overflow, the Level 2 BLAS routine ?tpsv is called. If the matrix A is singular ($A(j, j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $A^*x = 0$ is returned.

Input Parameters

<i>uplo</i>	CHARACTER*1.
	Specifies whether the matrix A is upper or lower triangular.
= 'U'	: upper triangular
= 'L'	: lower triangular
<i>trans</i>	CHARACTER*1.
	Specifies the operation applied to A .
= 'N'	: solve $A^*x = s^*b$ (no transpose)
= 'T'	: solve $A^T*x = s^*b$ (transpose)
= 'C'	: solve $A^H*x = s^*b$ (conjugate transpose)
<i>diag</i>	CHARACTER*1.
	Specifies whether the matrix A is unit triangular.
= 'N'	: non-unit triangular
= 'U'	: unit triangular
<i>normin</i>	CHARACTER*1.
	Specifies whether $cnorm$ is set.

= 'Y': *cnorm* contains the column norms on entry;
 = 'N': *cnorm* is not set on entry. On exit, the norms will be computed and stored in *cnorm*.

n

INTEGER. The order of the matrix *A*. $n \geq 0$.

ap

REAL for *slatps*
 DOUBLE PRECISION for *dlatps*
 COMPLEX for *clatps*
 DOUBLE COMPLEX for *zlatps*.

Array, DIMENSION ($n(n+1)/2$).

The upper or lower triangular matrix *A*, packed columnwise in a linear array. The *j*-th column of *A* is stored in the array *ap* as follows:

```
if uplo = 'U', ap(i + (j-1) $j/2$ ) = A(i, j) for  $1 \leq i \leq j$ ;  

if uplo = 'L', ap(i + (j-1)( $2n-j$ )/2) = A(i, j) for  $j \leq i \leq n$ .
```

x

REAL for *slatps* DOUBLE PRECISION for *dlatps*
 COMPLEX for *clatps*
 DOUBLE COMPLEX for *zlatps*.

Array, DIMENSION (*n*)

On entry, the right hand side *b* of the triangular system.

cnorm

REAL for *slatps/clatps*
 DOUBLE PRECISION for *dlatps/zlatps*.

Array, DIMENSION (*n*).

If *normin* = 'Y', *cnorm* is an input argument and *cnorm(j)* contains the norm of the off-diagonal part of the *j*-th column of *A*.

If *trans* = 'N', *cnorm(j)* must be greater than or equal to the infinity-norm, and if *trans* = 'T' or 'C', *cnorm(j)* must be greater than or equal to the 1-norm.

Output Parameters

x

On exit, *x* is overwritten by the solution vector *x*.

scale

REAL for *slatps/clatps*
 DOUBLE PRECISION for *dlatps/zlatps*.

The scaling factor *s* for the triangular system as described above.

If *scale* = 0, the matrix *A* is singular or badly scaled, and the vector *x* is an exact or approximate solution to $A^*x = 0$.

cnorm

If *normin* = 'N', *cnorm* is an output argument and *cnorm(j)* returns the 1-norm of the off-diagonal part of the *j*-th column of *A*.

info

INTEGER.

= 0: successful exit
 < 0: if *info* = -*k*, the *k*-th argument had an illegal value

?latrd

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

Syntax

```
call slatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call dlatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call clatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
call zlatrd( uplo, n, nb, a, lda, e, tau, w, ldw )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?latrd reduces *nb* rows and columns of a real symmetric or complex Hermitian matrix *A* to symmetric/Hermitian tridiagonal form by an orthogonal/unitary similarity transformation $Q^T * A * Q$ for real flavors, $Q^H * A * Q$ for complex flavors, and returns the matrices *V* and *W* which are needed to apply the transformation to the unreduced part of *A*.

If *uplo* = 'U', ?latrd reduces the last *nb* rows and columns of a matrix, of which the upper triangle is supplied;

if *uplo* = 'L', ?latrd reduces the first *nb* rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by ?sytrd/?hetrd.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix <i>A</i> is stored: = 'U': upper triangular = 'L': lower triangular
<i>n</i>	INTEGER. The order of the matrix <i>A</i> .
<i>nb</i>	INTEGER. The number of rows and columns to be reduced.
<i>a</i>	REAL for slatrd DOUBLE PRECISION for dlatrd COMPLEX for clatrd DOUBLE COMPLEX for zlatrd.

Array, DIMENSION (lda, n).

On entry, the symmetric/Hermitian matrix A

If $uplo = 'U'$, the leading n -by- n upper triangular part of a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

lda

INTEGER. The leading dimension of the array a . $lda \geq (1, n)$.

ldw

INTEGER.

The leading dimension of the output array w . $ldw \geq \max(1, n)$.

Output Parameters

a

On exit, if $uplo = 'U'$, the last nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of a ; the elements above the diagonal with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors;

if $uplo = 'L'$, the first nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of a ; the elements below the diagonal with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

e

REAL for `slatrd/clatrd`

DOUBLE PRECISION for `dlatrd/zlatrd`.

If $uplo = 'U'$, $e(n-nb:n-1)$ contains the superdiagonal elements of the last nb columns of the reduced matrix;

if $uplo = 'L'$, $e(1:nb)$ contains the subdiagonal elements of the first nb columns of the reduced matrix.

tau

REAL for `slatrd`

DOUBLE PRECISION for `dlatrd`

COMPLEX for `clatrd`

DOUBLE COMPLEX for `zlatrd`.

Array, DIMENSION (lda, n).

The scalar factors of the elementary reflectors, stored in $tau(n-nb:n-1)$ if $uplo = 'U'$, and in $tau(1:nb)$ if $uplo = 'L'$.

w

REAL for `slatrd`

DOUBLE PRECISION for `dlatrd`

COMPLEX for `clatrd`

DOUBLE COMPLEX for `zlatrd`.

Array, DIMENSION (ldw, n).

The n -by- nb matrix W required to update the unreduced part of A .

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n) * H(n-1) * \dots * H(n-nb+1)$$

Each $H(i)$ has the form

$$H(i) = I - tau * v * v'$$

where tau is a real/complex scalar, and v is a real/complex vector with $v(i:n) = 0$ and $v(i-1) = 1$; $v(1:i-1)$ is stored on exit in $a(1: i-1, i)$, and tau in $tau(i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form $H(i) = I - tau * v * v'$

where tau is a real/complex scalar, and v is a real/complex vector with $v(1: i) = 0$ and $v(i+1) = 1$; $v(i+1:n)$ is stored on exit in $a(i+1:n, i)$, and tau in $tau(i)$.

The elements of the vectors v together form the n -by- nb matrix V which is needed, with W , to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank-2k update of the form:

$$A := A - VW' - WV'.$$

The contents of a on exit are illustrated by the following examples with $n = 5$ and $nb = 2$:

if $uplo = 'U'$:

$$\begin{bmatrix} a & a & a & v_4 & v_4 \\ a & a & v_4 & v_5 & \\ a & 1 & v_5 & & \\ d & 1 & & & \\ d & & & & \end{bmatrix}$$

if $uplo = 'L'$

$$\begin{bmatrix} d & & & & \\ 1 & d & & & \\ v_1 & 1 & a & & \\ v_1 & v_2 & a & a & \\ v_1 & v_2 & a & a & a \end{bmatrix}$$

where d denotes a diagonal element of the reduced matrix, a denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining $H(i)$.

?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

```
call slatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call dlatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call clatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
call zlatrs( uplo, trans, diag, normin, n, a, lda, x, scale, cnorm, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine solves one of the triangular systems

$$A^*x = s^*b, \text{ or } A^T*x = s^*b, \text{ or } A^H*x = s^*b \text{ (for complex flavors)}$$

with scaling to prevent overflow. Here A is an upper or lower triangular matrix, A^T denotes the transpose of A , A^H denotes the conjugate transpose of A , x and b are n -element vectors, and s is a scaling factor, usually less than or equal to 1, chosen so that the components of x will be less than the overflow threshold. If the unscaled problem will not cause overflow, the Level 2 BLAS routine ?trsv is called. If the matrix A is singular ($A(j, j) = 0$ for some j), then s is set to 0 and a non-trivial solution to $A^*x = 0$ is returned.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular = 'L': Lower triangular
<i>trans</i>	CHARACTER*1. Specifies the operation applied to A . = 'N': solve $A^*x = s^*b$ (no transpose) = 'T': solve $A^T*x = s^*b$ (transpose) = 'C': solve $A^H*x = s^*b$ (conjugate transpose)
<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix A is unit triangular. = 'N': non-unit triangular = 'U': unit triangular
<i>normin</i>	CHARACTER*1. Specifies whether <i>cnorm</i> has been set or not. = 'Y': <i>cnorm</i> contains the column norms on entry; = 'N': <i>cnorm</i> is not set on entry. On exit, the norms will be computed and stored in <i>cnorm</i> .
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$
<i>a</i>	REAL for slatrs DOUBLE PRECISION for dlatrs COMPLEX for clatrs DOUBLE COMPLEX for zlatrs. Array, DIMENSION (<i>lda</i> , <i>n</i>). Contains the triangular matrix A .

If $uplo = 'U'$, the leading n -by- n upper triangular part of the array a contains the upper triangular matrix, and the strictly lower triangular part of A is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of the array a contains the lower triangular matrix, and the strictly upper triangular part of A is not referenced.

If $diag = 'U'$, the diagonal elements of A are also not referenced and are assumed to be 1.

lda INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

x REAL for slatrs

DOUBLE PRECISION for dlatrs

COMPLEX for clatrs

DOUBLE COMPLEX for zlatrs .

Array, DIMENSION (n).

On entry, the right hand side b of the triangular system.

$cnorm$ REAL for $\text{slatrs}/\text{clatrs}$

DOUBLE PRECISION for $\text{dlatrs}/\text{zlatrs}$.

Array, DIMENSION (n).

If $normin = 'Y'$, $cnorm$ is an input argument and $cnorm(j)$ contains the norm of the off-diagonal part of the j -th column of A .

If $trans = 'N'$, $cnorm(j)$ must be greater than or equal to the infinity-norm, and if $trans = 'T'$ or ' C' , $cnorm(j)$ must be greater than or equal to the 1-norm.

Output Parameters

x On exit, x is overwritten by the solution vector x .

$scale$ REAL for $\text{slatrs}/\text{clatrs}$

DOUBLE PRECISION for $\text{dlatrs}/\text{zlatrs}$.

Array, DIMENSION (lda, n). The scaling factor s for the triangular system as described above.

If $scale = 0$, the matrix A is singular or badly scaled, and the vector x is an exact or approximate solution to $A^*x = 0$.

$cnorm$ If $normin = 'N'$, $cnorm$ is an output argument and $cnorm(j)$ returns the 1-norm of the off-diagonal part of the j -th column of A .

$info$ INTEGER.

= 0: successful exit

< 0: if $info = -k$, the k -th argument had an illegal value

Application Notes

A rough bound on x is computed; if that is less than overflow, `?trsv` is called, otherwise, specific code is used which checks for possible overflow or divide-by-zero at every operation.

A columnwise scheme is used for solving $Ax = b$. The basic algorithm if A is lower triangular is

```

x[1:n] := b[1:n]
for j = 1, ..., n
  x(j) := x(j) / A(j,j)
  x[j+1:n] := x[j+1:n] - x(j)*a[j+1:n,j]
end

```

Define bounds on the components of x after j iterations of the loop:

```

M(j) = bound on x[1:j]
G(j) = bound on x[j+1:n]
Initially, let M(0) = 0 and G(0) = max{x(i), i=1,...,n}.

```

Then for iteration $j+1$ we have

$$\begin{aligned} M(j+1) &\leq G(j) / |a(j+1, j+1)| \\ G(j+1) &\leq G(j) + M(j+1) * |a[j+2:n, j+1]| \\ &\leq G(j) (1 + cnorm(j+1) / |a(j+1, j+1)|), \end{aligned}$$

where $cnorm(j+1)$ is greater than or equal to the infinity-norm of column $j+1$ of a , not counting the diagonal. Hence

$$G(j) \leq G(0) \prod_{1 \leq i \leq j} (1 + cnorm(i)/|A(i,i)|)$$

and

$$|x(j)| \leq (G(0)/|A(j,j)|) \prod_{1 \leq i \leq j} (1 + cnorm(i)/|A(i,i)|)$$

Since $|x(j)| \leq M(j)$, we use the Level 2 BLAS routine `?trsv` if the reciprocal of the largest $M(j)$, $j=1,\dots,n$, is larger than $\max(\text{underflow}, 1/\text{overflow})$.

The bound on $x(j)$ is also used to determine when a step in the columnwise method can be performed without fear of overflow. If the computed bound is greater than a large constant, x is scaled to prevent overflow, but if the bound overflows, x is set to 0, $x(j)$ to 1, and scale to 0, and a non-trivial solution to $Ax = 0$ is found.

Similarly, a row-wise scheme is used to solve $A^T x = b$ or $A^H x = b$. The basic algorithm for A upper triangular is

```

for j = 1, ..., n
  x(j) := (b(j) - A[1:j-1,j]' x[1:j-1]) / A(j,j)
end

```

We simultaneously compute two bounds

$G(j) = \text{bound on } (b(i) - A[1:i-1, i]' * x[1:i-1]), 1 \leq i \leq j$

$M(j) = \text{bound on } x(i), 1 \leq i \leq j$

The initial values are $G(0) = 0, M(0) = \max\{b(i), i=1,\dots,n\}$, and we add the constraint $G(j) \geq G(j-1)$ and $M(j) \geq M(j-1)$ for $j \geq 1$.

Then the bound on $x(j)$ is

$$M(j) \leq M(j-1) * (1 + cnorm(j)) / |A(j,j)|$$

$$\leq M(0) \prod_{1 \leq i \leq j} (1 + cnorm(i)/|A(i,i)|)$$

and we can safely call ?trsv if $1/M(n)$ and $1/G(n)$ are both greater than $\max(\text{underflow}, 1/\text{overflow})$.

?latrz

Factors an upper trapezoidal matrix by means of orthogonal/unitary transformations.

Syntax

```
call slatrz( m, n, l, a, lda, tau, work )
call dlatrz( m, n, l, a, lda, tau, work )
call clatrz( m, n, l, a, lda, tau, work )
call zlatrz( m, n, l, a, lda, tau, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?latrz factors the m -by- $(m+l)$ real/complex upper trapezoidal matrix

$$[A1 \ A2] = [A(1:m, 1:m) \ A(1: m, n-l+1:n)]$$

as $(R \ 0) * Z$, by means of orthogonal/unitary transformations. Z is an $(m+l)$ -by- $(m+l)$ orthogonal/unitary matrix and R and $A1$ are m -by- m upper triangular matrices.

Input Parameters

m	INTEGER. The number of rows of the matrix A . $m \geq 0$.
n	INTEGER. The number of columns of the matrix A . $n \geq 0$.
l	INTEGER. The number of columns of the matrix A containing the meaningful part of the Householder vectors. $n-m \geq l \geq 0$.
a	REAL for slatrz DOUBLE PRECISION for dlatrz COMPLEX for clatrz

DOUBLE COMPLEX for zlatrz.

Array, DIMENSION (*lda, n*).

On entry, the leading *m*-by-*n* upper trapezoidal part of the array *a* must contain the matrix to be factorized.

lda INTEGER. The leading dimension of the array *a*. *lda* $\geq \max(1, m)$.

work REAL for slatrz

DOUBLE PRECISION for dlatrz

COMPLEX for clatrz

DOUBLE COMPLEX for zlatrz.

Workspace array, DIMENSION (*m*).

Output Parameters

a On exit, the leading *m*-by-*m* upper triangular part of *a* contains the upper triangular matrix *R*, and elements *n-l+1* to *n* of the first *m* rows of *a*, with the array *tau*, represent the orthogonal/unitary matrix *Z* as a product of *m* elementary reflectors.

tau REAL for slatrz

DOUBLE PRECISION for dlatrz

COMPLEX for clatrz

DOUBLE COMPLEX for zlatrz.

Array, DIMENSION (*m*).

The scalar factors of the elementary reflectors.

Application Notes

The factorization is obtained by Householder's method. The *k*-th transformation matrix, *z(k)*, which is used to introduce zeros into the (*m - k + 1*)-th row of *A*, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix}$$

where for real flavors

$$T(k) = I - t \alpha u^* u(k)^* T(k)^T, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ z(k) \end{bmatrix}$$

and for complex flavors

$$T(k) = I - \tau u u^H T(k)^H, \quad u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

τ is a scalar and $z(k)$ is an l -element vector. τ and $z(k)$ are chosen to annihilate the elements of the k -th row of A_2 .

The scalar τ is returned in the k -th element of τ and the vector $u(k)$ in the k -th row of A_2 , such that the elements of $z(k)$ are in $a(k, l+1), \dots, a(k, n)$.

The elements of r are returned in the upper triangular part of A_1 .

Z is given by

$$Z = Z(1) * Z(2) * \dots * Z(m).$$

?laau2

Computes the product $U^* U^T (U^* U^H)$ or $L^T * L (L^H * L)$, where U and L are upper or lower triangular matrices (unblocked algorithm).

Syntax

```
call slauu2( uplo, n, a, lda, info )
call dlauu2( uplo, n, a, lda, info )
call clauu2( uplo, n, a, lda, info )
call zlauu2( uplo, n, a, lda, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?laau2 computes the product $U^* U^T$ or $L^T * L$ for real flavors, and $U^* U^H$ or $L^H * L$ for complex flavors. Here the triangular factor U or L is stored in the upper or lower triangular part of the array a .

If $uplo = 'U'$ or ' u ', then the upper triangle of the result is stored, overwriting the factor U in A .

If $uplo = 'L'$ or ' l ', then the lower triangle of the result is stored, overwriting the factor L in A .

This is the unblocked form of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

$uplo$	CHARACTER*1.
	Specifies whether the triangular factor stored in the array a is upper or lower triangular:
= ' U '	: Upper triangular
= ' L '	: Lower triangular

n INTEGER. The order of the triangular factor U or L . $n \geq 0$.

a REAL for `slauu2`
 DOUBLE PRECISION for `dlauu2`
 COMPLEX for `clauu2`
 DOUBLE COMPLEX for `zlauu2`.
 Array, DIMENSION (*lda, n*). On entry, the triangular factor *U* or *L*.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a On exit,
 if *uplo* = 'U', then the upper triangle of *a* is overwritten with the upper triangle of the product $U^* U^T (U^* U^H)$;
 if *uplo* = 'L', then the lower triangle of *a* is overwritten with the lower triangle of the product $L^T * L (L^H * L)$.

info INTEGER.
 = 0: successful exit
 < 0: if *info* = $-k$, the *k*-th argument had an illegal value

?lauum

Computes the product $U^* U^T (U^* U^H)$ or $L^T * L (L^H * L)$, where *U* and *L* are upper or lower triangular matrices (blocked algorithm).

Syntax

```
call slauum( uplo, n, a, lda, info )
call dlauum( uplo, n, a, lda, info )
call clauum( uplo, n, a, lda, info )
call zlauum( uplo, n, a, lda, info )
```

C:

```
lapack_int LAPACKE_<?>lauum (int matrix_layout, char uplo, lapack_int n, <datatype> * a, lapack_int lda);
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine `?lauum` computes the product $U^* U^T$ or $L^T * L$ for real flavors, and $U^* U^H$ or $L^H * L$ for complex flavors. Here the triangular factor *U* or *L* is stored in the upper or lower triangular part of the array *a*.

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor *U* in *A*.

If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor *L* in *A*.

This is the blocked form of the algorithm, calling [BLAS Level 3 Routines](#).

Input Parameters

The data types are given for the Fortran interface.

<i>uplo</i>	CHARACTER*1.
	Specifies whether the triangular factor stored in the array <i>a</i> is upper or lower triangular:
= 'U'	: Upper triangular
= 'L'	: Lower triangular
<i>n</i>	INTEGER. The order of the triangular factor <i>U</i> or <i>L</i> . $n \geq 0$.
<i>a</i>	REAL for <i>slauum</i> DOUBLE PRECISION for <i>dlauum</i> COMPLEX for <i>clauum</i> DOUBLE COMPLEX for <i>zlaum</i> . Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the triangular factor <i>U</i> or <i>L</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.

Output Parameters

<i>a</i>	On exit, if <i>uplo</i> = 'U', then the upper triangle of <i>a</i> is overwritten with the upper triangle of the product $U^* U^T (U^* U^H)$; if <i>uplo</i> = 'L', then the lower triangle of <i>a</i> is overwritten with the lower triangle of the product $L^T * L (L^H * L)$.
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = - <i>k</i> , the <i>k</i> -th argument had an illegal value

?orbdb1/?unbdb1

Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

Syntax

FORTRAN 77:

```
call sorbdb1( m, p, q, x11, idx11, x21, idx21, theta, phi, taup1, taup2, tauq1, work,  
lwork, info )  
  
call dorbdb1( m, p, q, x11, idx11, x21, idx21, theta, phi, taup1, taup2, tauq1, work,  
lwork, info )  
  
call cunbdb1( m, p, q, x11, idx11, x21, idx21, theta, phi, taup1, taup2, tauq1, work,  
lwork, info )
```

```
call zunbdb1( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routines ?orbdb1/?unbdb1 simultaneously bidiagonalize the blocks of a tall and skinny matrix X with orthonormal columns:

$$\begin{bmatrix} x_{11} \\ x_{21} \end{bmatrix} = \begin{bmatrix} p_1 & | & p_2 \end{bmatrix} \begin{bmatrix} b_{11} \\ 0 \\ b_{21} \\ 0 \end{bmatrix} q_1^T$$

The size of x_{11} is p by q , and x_{21} is $(m - p)$ by q . q must not be larger than p , $m - p$, or $m - q$.

Tall and Skinny Matrix Routines

$q \leq \min(p, m - p, m - q)$?orbdb1/?unbdb1
$p \leq \min(q, m - p, m - q)$?orbdb2/?unbdb2
$m - p \leq \min(p, q, m - q)$?orbdb3/?unbdb3
$m - q \leq \min(p, q, m - p)$?orbdb4/?unbdb4

The orthogonal/unitary matrices p_1 , p_2 , and q_1 are p -by- p , $(m-p)$ -by- $(m-p)$, $(m-q)$ -by- $(m-q)$, respectively.

p_1 , p_2 , and q_1 are represented as products of elementary reflectors. See the description of ?orcsd2by1/?uncsd2by1 for details on generating p_1 , p_2 , and q_1 using ?orgqr and ?orglq.

The upper-bidiagonal matrices b_{11} and b_{21} of size q by q are represented implicitly by angles $\theta(1), \dots, \theta(q)$ and $\phi(1), \dots, \phi(q-1)$. Every entry in each bidiagonal band is a product of a sine or cosine of θ with a sine or cosine of ϕ . See [Sutton09] or the description of ?orcsd/?uncsd for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m INTEGER. The number of rows in x_{11} plus the number of rows in x_{21} .

p INTEGER. The number of rows in x_{11} . $0 \leq p \leq m$.

q INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq \min(p, m - p, m - q)$.

x11 REAL for sorbdb1

DOUBLE PRECISION for dorbdb1

COMPLEX for cunbdb1

DOUBLE COMPLEX for zunbdb1

Array, DIMENSION (l_{dx11}, q).

On entry, the top block of the orthogonal/unitary matrix to be reduced.

l_{dx11}

INTEGER. The leading dimension of the array X_{11} . $l_{dx11} \geq p$.

x_{21}

REAL for `sorbdb1`

DOUBLE PRECISION for `dorbdb1`

COMPLEX for `cunbdb1`

DOUBLE COMPLEX for `zunbdb1`

Array, DIMENSION (l_{dx21}, q).

On entry, the bottom block of the orthogonal/unitary matrix to be reduced.

l_{dx21}

INTEGER. The leading dimension of the array X_{21} . $l_{dx21} \geq m-p$.

$work$

REAL for `sorbdb1`

DOUBLE PRECISION for `dorbdb1`

COMPLEX for `cunbdb1`

DOUBLE COMPLEX for `zunbdb1`

Workspace array, DIMENSION (l_{work}).

l_{work}

INTEGER. The size of the $work$ array. $l_{work} \geq m-q$

If $l_{work} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to l_{work} is issued by `xerbla`.

Output Parameters

x_{11}

The columns of `tril(x11)` specify reflectors for p_1 and the rows of `triu(x11, 1)` specify reflectors for q_1 , where `tril(A)` denotes the lower triangle of A , and `triu(A)` denotes the upper triangle of A .

x_{21}

On exit, the columns of `tril(x21)` specify the reflectors for p_2

$theta$

REAL for `sorbdb1`

DOUBLE PRECISION for `dorbdb1`

COMPLEX for `cunbdb1`

DOUBLE COMPLEX for `zunbdb1`

Array, DIMENSION (q). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles $theta$ and phi . See the Description section for details.

phi

REAL for `sorbdb1`

DOUBLE PRECISION for `dorbdb1`

COMPLEX for `cunbdb1`

DOUBLE COMPLEX for `zunbdb1`

Array, DIMENSION ($q-1$). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

taup1

REAL for sorbdb1
 DOUBLE PRECISION for dorbdb1
 COMPLEX for cunbdb1
 DOUBLE COMPLEX for zunbdb1
Array, DIMENSION (p).

Scalar factors of the elementary reflectors that define p_1 .

taup2

REAL for sorbdb1
 DOUBLE PRECISION for dorbdb1
 COMPLEX for cunbdb1
 DOUBLE COMPLEX for zunbdb1
Array, DIMENSION ($m-p$).

Scalar factors of the elementary reflectors that define p_2 .

taug1

REAL for sorbdb1
 DOUBLE PRECISION for dorbdb1
 COMPLEX for cunbdb1
 DOUBLE COMPLEX for zunbdb1
Array, DIMENSION (q).

Scalar factors of the elementary reflectors that define q_1 .

info

INTEGER.
 = 0: successful exit
 < 0: if *info* = $-i$, the i -th argument has an illegal value.

See Also

[?orcsd/?uncsd](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

[?orcsd2by1/?uncsd2by1](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

[?orbdb2/?unbdb2](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb3/?unbdb3](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb4/?unbdb4](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb5/?unbdb5](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[?orbdb6/?unbdb6](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[xerbla](#)

?orbdb2/?unbdb2

Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

Syntax

FORTRAN 77:

```
call sorbdb2( m, p, q, x11, idx11, x21, idx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )

call dorbdb2( m, p, q, x11, idx11, x21, idx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )

call cunbdb2( m, p, q, x11, idx11, x21, idx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )

call zunbdb2( m, p, q, x11, idx11, x21, idx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routines ?orbdb2/?unbdb2 simultaneously bidiagonalize the blocks of a tall and skinny matrix X with orthonormal columns:

$$\begin{bmatrix} \frac{x_{11}}{x_{21}} \end{bmatrix} = \begin{bmatrix} p_1 & | & p_2 \end{bmatrix} \begin{bmatrix} b_{11} \\ 0 \\ \frac{b_{21}}{b_{11}} \\ 0 \end{bmatrix} q_1^\top$$

The size of x_{11} is p by q , and x_{12} is $(m - p)$ by q . q must not be larger than p , $m-p$, or $m-q$.

Tall and Skinny Matrix Routines

$q \leq \min(p, m - p, m - q)$?orbdb1/?unbdb1
$p \leq \min(q, m - p, m - q)$?orbdb2/?unbdb2
$m - p \leq \min(p, q, m - q)$?orbdb3/?unbdb3
$m - q \leq \min(p, q, m - p)$?orbdb4/?unbdb4

The orthogonal/unitary matrices p_1 , p_2 , and q_1 are p -by- p , $(m-p)$ -by- $(m-p)$, $(m-q)$ -by- $(m-q)$, respectively.

p_1 , p_2 , and q_1 are represented as products of elementary reflectors. See the description of ?orcsd2by1/?uncsd2by1 for details on generating p_1 , p_2 , and q_1 using ?orgqr and ?orglq.

The upper-bidiagonal matrices b_{11} and b_{21} of size p by p are represented implicitly by angles $\theta(1), \dots, \theta(q)$ and $\phi(1), \dots, \phi(q-1)$. Every entry in each bidiagonal band is a product of a sine or cosine of θ with a sine or cosine of ϕ . See [Sutton09] or the description of ?orcsd/?uncsd for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m

INTEGER. The number of rows in x_{11} plus the number of rows in x_{21} .

p INTEGER. The number of rows in x_{11} . $0 \leq p \leq \min(q, m-p, m-q)$.

q INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq m$.

x₁₁ REAL for sorbdb2
 DOUBLE PRECISION for dorbdb2
 COMPLEX for cunbdb2
 DOUBLE COMPLEX for zunbdb2
Array, DIMENSION (*lidx11,q*).
 On entry, the top block of the orthogonal/unitary matrix to be reduced.

lidx11 INTEGER. The leading dimension of the array X_{11} . $lidx11 \geq p$.

x₂₁ REAL for sorbdb2
 DOUBLE PRECISION for dorbdb2
 COMPLEX for cunbdb2
 DOUBLE COMPLEX for zunbdb2
Array, DIMENSION (*lidx21,q*).
 On entry, the bottom block of the orthogonal/unitary matrix to be reduced.

lidx21 INTEGER. The leading dimension of the array X_{21} . $lidx21 \geq m-p$.

work REAL for sorbdb2
 DOUBLE PRECISION for dorbdb2
 COMPLEX for cunbdb2
 DOUBLE COMPLEX for zunbdb2
Workspace array, DIMENSION (*lwork*).

lwork INTEGER. The size of the *work* array. $lwork \geq m-q$
 If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

x₁₁ On exit: the columns of $\text{tril}(x_{11})$ specify reflectors for p_1 and the rows of $\text{triu}(x_{11}, 1)$ specify reflectors for q_1 .

x₂₁ On exit, the columns of $\text{tril}(x_{21})$ specify the reflectors for p_2

theta REAL for sorbdb2
 DOUBLE PRECISION for dorbdb2
 COMPLEX for cunbdb2
 DOUBLE COMPLEX for zunbdb2

Array, DIMENSION (q). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

phi

```
REAL for sorbdb2
DOUBLE PRECISION for dorbdb2
COMPLEX for cunbdb2
DOUBLE COMPLEX for zunbdb2
```

Array, DIMENSION (q-1). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

taup1

```
REAL for sorbdb2
DOUBLE PRECISION for dorbdb2
COMPLEX for cunbdb2
DOUBLE COMPLEX for zunbdb2
```

Array, DIMENSION (p).

Scalar factors of the elementary reflectors that define p_1 .

taup2

```
REAL for sorbdb2
DOUBLE PRECISION for dorbdb2
COMPLEX for cunbdb2
DOUBLE COMPLEX for zunbdb2
```

Array, DIMENSION (m-p).

Scalar factors of the elementary reflectors that define p_2 .

tauq1

```
REAL for sorbdb2
DOUBLE PRECISION for dorbdb2
COMPLEX for cunbdb2
DOUBLE COMPLEX for zunbdb2
```

Array, DIMENSION (q).

Scalar factors of the elementary reflectors that define q_1 .

info

```
INTEGER.
= 0: successful exit
< 0: if info = -i, the i-th argument has an illegal value.
```

See Also

?orcsd/?uncsd

?orcsd2by1/?uncsd2by1 Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

?orbdb1/?unbdb1 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb3/?unbdb3 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb4/?unbdb4](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb5/?unbdb5](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[?orbdb6/?unbdb6](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.
xerbla

?orbdb3/?unbdb3

Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

Syntax

FORTRAN 77:

```
call sorbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
call dorbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
call cunbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
call zunbdb3( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1, work,
lwork, info )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routines [?orbdb3/?unbdb3](#) simultaneously bidiagonalize the blocks of a tall and skinny matrix X with orthonormal columns:

$$\begin{bmatrix} \frac{x_{11}}{x_{21}} \end{bmatrix} = \left[\begin{array}{c|c} p_1 & \\ \hline & p_2 \end{array} \right] \begin{bmatrix} b_{11} \\ 0 \\ \hline b_{21} \\ 0 \end{bmatrix} q_1^T$$

The size of x_{11} is p by q , and x_{12} is $(m - p)$ by q . $m - p$ must not be larger than p , q , or $m - q$.

Tall and Skinny Matrix Routines

$q \leq \min(p, m - p, m - q)$?orbdb1/?unbdb1
$p \leq \min(q, m - p, m - q)$?orbdb2/?unbdb2
$m - p \leq \min(p, q, m - q)$?orbdb3/?unbdb3
$m - q \leq \min(p, q, m - p)$?orbdb4/?unbdb4

The orthogonal/unitary matrices p_1 , p_2 , and q_1 are p -by- p , $(m-p)$ -by- $(m-p)$, $(m-q)$ -by- $(m-q)$, respectively.

p_1 , p_2 , and q_1 are represented as products of elementary reflectors. See the description of [?orcisd2by1/?uncsd2by1](#) for details on generating p_1 , p_2 , and q_1 using [?orgqr](#) and [?orglq](#).

The upper-bidiagonal matrices b_{11} and b_{12} of size $(m-p)$ by $(m-p)$ are represented implicitly by angles $\theta(1), \dots, \theta(q)$ and $\phi(1), \dots, \phi(q-1)$. Every entry in each bidiagonal band is a product of a sine or cosine of θ with a sine or cosine of ϕ . See [Sutton09] or the description of ?orcsd/?uncsd for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m	INTEGER. The number of rows in x_{11} plus the number of rows in x_{21} .
p	INTEGER. The number of rows in x_{11} . $0 \leq p \leq m$, $m-p \leq \min(p, q, m-q)$.
q	INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq m$.
x_{11}	<p>REAL for sorbdb3 DOUBLE PRECISION for dorbdb3 COMPLEX for cunbdb3 DOUBLE COMPLEX for zunbdb3 Array, DIMENSION (l_{dx11}, q). On entry, the top block of the orthogonal/unitary matrix to be reduced.</p>
l_{dx11}	INTEGER. The leading dimension of the array X_{11} . $l_{dx11} \geq p$.
x_{21}	<p>REAL for sorbdb3 DOUBLE PRECISION for dorbdb3 COMPLEX for cunbdb3 DOUBLE COMPLEX for zunbdb3 Array, DIMENSION (l_{dx21}, q). On entry, the bottom block of the orthogonal/unitary matrix to be reduced.</p>
l_{dx21}	INTEGER. The leading dimension of the array X_{21} . $l_{dx21} \geq m-p$.
$work$	<p>REAL for sorbdb3 DOUBLE PRECISION for dorbdb3 COMPLEX for cunbdb3 DOUBLE COMPLEX for zunbdb3 Workspace array, DIMENSION (l_{work}).</p>
l_{work}	<p>INTEGER. The size of the $work$ array. $l_{work} \geq m-q$ If $l_{work} = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to l_{work} is issued by xerbla.</p>

Output Parameters

<i>x11</i>	On exit: the columns of <code>tril(x11)</code> specify reflectors for p_1 and the rows of <code>triu(x11, 1)</code> specify reflectors for q_1 .
<i>x21</i>	On exit, the columns of <code>tril(x21)</code> specify the reflectors for p_2
<i>theta</i>	<p>REAL for <code>sorbdb3</code></p> <p>DOUBLE PRECISION for <code>dorbdb3</code></p> <p>COMPLEX for <code>cunbdb3</code></p> <p>DOUBLE COMPLEX for <code>zunbdb3</code></p> <p>Array, DIMENSION (q). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles <i>theta</i> and <i>phi</i>. See the Description section for details.</p>
<i>phi</i>	<p>REAL for <code>sorbdb3</code></p> <p>DOUBLE PRECISION for <code>dorbdb3</code></p> <p>COMPLEX for <code>cunbdb3</code></p> <p>DOUBLE COMPLEX for <code>zunbdb3</code></p> <p>Array, DIMENSION ($q-1$). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles <i>theta</i> and <i>phi</i>. See the Description section for details.</p>
<i>taup1</i>	<p>REAL for <code>sorbdb3</code></p> <p>DOUBLE PRECISION for <code>dorbdb3</code></p> <p>COMPLEX for <code>cunbdb3</code></p> <p>DOUBLE COMPLEX for <code>zunbdb3</code></p> <p>Array, DIMENSION (p).</p> <p>Scalar factors of the elementary reflectors that define p_1.</p>
<i>taup2</i>	<p>REAL for <code>sorbdb3</code></p> <p>DOUBLE PRECISION for <code>dorbdb3</code></p> <p>COMPLEX for <code>cunbdb3</code></p> <p>DOUBLE COMPLEX for <code>zunbdb3</code></p> <p>Array, DIMENSION ($m-p$).</p> <p>Scalar factors of the elementary reflectors that define p_2.</p>
<i>tauq1</i>	<p>REAL for <code>sorbdb3</code></p> <p>DOUBLE PRECISION for <code>dorbdb3</code></p> <p>COMPLEX for <code>cunbdb3</code></p> <p>DOUBLE COMPLEX for <code>zunbdb3</code></p> <p>Array, DIMENSION (q).</p> <p>Scalar factors of the elementary reflectors that define q_1.</p>
<i>info</i>	INTEGER.

= 0: successful exit
< 0: if *info* = -*i*, the *i*-th argument has an illegal value.

See Also

?orcsd/?uncsd

?orcsd2by1/?uncsd2by1 Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

?orbdb1/?unbdb1 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb2/?unbdb2 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb4/?unbdb4 Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

?orbdb5/?unbdb5 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

?orbdb6/?unbdb6 Orthogonalizes a column vector with respect to the orthonormal basis matrix.

xerbla

?orbdb4/?unbdb4

Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

Syntax

FORTRAN 77:

```
call sorbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1,
phantom, work, lwork, info )
call dorbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1,
phantom, work, lwork, info )
call cunbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1,
phantom, work, lwork, info )
call zunbdb4( m, p, q, x11, ldx11, x21, ldx21, theta, phi, taup1, taup2, tauq1,
phantom, work, lwork, info )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routines ?orbdb4/?unbdb4 simultaneously bidiagonalize the blocks of a tall and skinny matrix *X* with orthonormal columns:

$$\begin{bmatrix} \frac{x_{11}}{x_{21}} \end{bmatrix} = \begin{bmatrix} p_1 & | & p_2 \end{bmatrix} \begin{bmatrix} b_{11} \\ 0 \\ \frac{b_{21}}{b_{11}} \\ 0 \end{bmatrix} q_1^T$$

The size of *x*₁₁ is *p* by *q*, and *x*₁₂ is (*m* - *p*) by *q*. *m* - *q* must not be larger than *q*, *p*, or *m* - *p*.

Tall and Skinny Matrix Routines

$q \leq \min(p, m - p, m - q)$?orbdb1/?unbdb1
$p \leq \min(q, m - p, m - q)$?orbdb2/?unbdb2
$m - p \leq \min(p, q, m - q)$?orbdb3/?unbdb3
$m - q \leq \min(p, q, m - p)$?orbdb4/?unbdb4

The orthogonal/unitary matrices p_1 , p_2 , and q_1 are p -by- p , $(m-p)$ -by- $(m-p)$, $(m-q)$ -by- $(m-q)$, respectively.

p_1 , p_2 , and q_1 are represented as products of elementary reflectors. See the description of ?orcsd2by1/?uncsd2by1 for details on generating p_1 , p_2 , and q_1 using ?orgqr and ?orglq.

The upper-bidiagonal matrices b_{11} and b_{12} of size $(m-q)$ by $(m-q)$ are represented implicitly by angles $\text{theta}(1), \dots, \text{theta}(q)$ and $\text{phi}(1), \dots, \text{phi}(q-1)$. Every entry in each bidiagonal band is a product of a sine or cosine of theta with a sine or cosine of phi . See [Sutton09] or the description of ?orcsd/?uncsd for details.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

m	INTEGER. The number of rows in x_{11} plus the number of rows in x_{21} .
p	INTEGER. The number of rows in x_{11} . $0 \leq p \leq m$.
q	INTEGER. The number of columns in x_{11} and x_{21} . $0 \leq q \leq m$ and $0 \leq m-q \leq \min(p, m-p, q)$.
x_{11}	REAL for sorbdb4 DOUBLE PRECISION for dorbdb4 COMPLEX for cunbdb4 DOUBLE COMPLEX for zunbdb4 Array, DIMENSION (l_{dx11}, q). On entry, the top block of the orthogonal/unitary matrix to be reduced.
l_{dx11}	INTEGER. The leading dimension of the array X_{11} . $l_{dx11} \geq p$.
x_{21}	REAL for sorbdb4 DOUBLE PRECISION for dorbdb4 COMPLEX for cunbdb4 DOUBLE COMPLEX for zunbdb4 Array, DIMENSION (l_{dx21}, q). On entry, the bottom block of the orthogonal/unitary matrix to be reduced.
l_{dx21}	INTEGER. The leading dimension of the array X_{21} . $l_{dx21} \geq m-p$.
$work$	REAL for sorbdb4 DOUBLE PRECISION for dorbdb4

COMPLEX for cunbdb4
DOUBLE COMPLEX for zunbdb4
Workspace array, DIMENSION (*lwork*).

lwork **INTEGER.** The size of the *work* array. $lwork \geq m-q$

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by xerbla.

Output Parameters

x11 **On exit:** the columns of `tril(x11)` specify reflectors for p_1 and the rows of `triu(x11,1)` specify reflectors for q_1 .

x21 **On exit,** the columns of `tril(x21)` specify the reflectors for p_2

theta **REAL for sorbdb4**

DOUBLE PRECISION for dorbdb4

COMPLEX for cunbdb4

DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (q). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

phi **REAL for sorbdb4**

DOUBLE PRECISION for dorbdb4

COMPLEX for cunbdb4

DOUBLE COMPLEX for zunbdb4

Array, DIMENSION ($q-1$). The entries of bidiagonal blocks b_{11} and b_{21} can be computed from the angles *theta* and *phi*. See the Description section for details.

taup1 **REAL for sorbdb4**

DOUBLE PRECISION for dorbdb4

COMPLEX for cunbdb4

DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (p).

Scalar factors of the elementary reflectors that define p_1 .

taup2 **REAL for sorbdb4**

DOUBLE PRECISION for dorbdb4

COMPLEX for cunbdb4

DOUBLE COMPLEX for zunbdb4

Array, DIMENSION ($m-p$).

Scalar factors of the elementary reflectors that define p_2 .

taug1

REAL for sorbdb4
 DOUBLE PRECISION for dorbdb4
 COMPLEX for cunbdb4
 DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (q).

Scalar factors of the elementary reflectors that define q_1 .

phantom

REAL for sorbdb4
 DOUBLE PRECISION for dorbdb4
 COMPLEX for cunbdb4
 DOUBLE COMPLEX for zunbdb4

Array, DIMENSION (m).

The routine computes an m -by-1 column vector y that is orthogonal to the columns of $[x_{11}; x_{21}]$. $\text{phantom}(1:p)$ and $\text{phantom}(p+1:m)$ contain Householder vectors for $y(1:p)$ and $y(p+1:m)$, respectively.

info

INTEGER.
 = 0: successful exit
 < 0: if $\text{info} = -i$, the i -th argument has an illegal value.

See Also

[?orcsd/?uncsd](#)

[?orcsd2by1/?uncsd2by1](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

[?orbdb1/?unbdb1](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb2/?unbdb2](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb3/?unbdb3](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb5/?unbdb5](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[?orbdb6/?unbdb6](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.

[xerbla](#)

?orbdb5/?unbdb5

Orthogonalizes a column vector with respect to the orthonormal basis matrix.

Syntax

FORTRAN 77:

```
call sorbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call dorbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call cunbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call zunbdb5( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The ?orbdb5/?unbdb5 routines orthogonalize the column vector

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

with respect to the columns of

$$q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}$$

The columns of Q must be orthonormal.

If the projection is zero according to Kahan's "twice is enough" criterion, then some other vector from the orthogonal complement is returned. This vector is chosen in an arbitrary but deterministic way.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m1</i>	INTEGER	The dimension of <i>x1</i> and the number of rows in <i>q1</i> . $0 \leq m1$.
<i>m2</i>	INTEGER	The dimension of <i>x2</i> and the number of rows in <i>q2</i> . $0 \leq m2$.
<i>n</i>	INTEGER	The number of columns in <i>q1</i> and <i>q2</i> . $0 \leq n$.
<i>x1</i>	REAL for sorbdb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5 COMPLEX*16 for zundb5 Array of size <i>m1</i> .	The top part of the vector to be orthogonalized.
<i>incx1</i>	INTEGER	Increment for entries of <i>x1</i> .
<i>x2</i>	REAL for sorbdb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5	

COMPLEX*16 **for** zunbdb5
Array of size $m2$.
The bottom part of the vector to be orthogonalized.

incx2 INTEGER
Increment for entries of $x2$.

q1 REAL **for** sordb5
DOUBLE PRECISION **for** dordb5
COMPLEX **for** cundb5
COMPLEX*16 **for** zunbdb5
Array of size ($ldq1, n$).
The top part of the orthonormal basis matrix.

ldq1 INTEGER
The leading dimension of $q1$. $ldq1 \geq m1$.

q2 REAL **for** sordb5
DOUBLE PRECISION **for** dordb5
COMPLEX **for** cundb5
COMPLEX*16 **for** zunbdb5
Array of size ($ldq2, n$).
The bottom part of the orthonormal basis matrix.

ldq2 INTEGER
The leading dimension of $q2$. $ldq2 \geq m2$.

work REAL **for** sordb5
DOUBLE PRECISION **for** dordb5
COMPLEX **for** cundb5
COMPLEX*16 **for** zunbdb5
Workspace array of size $lwork$.

lwork INTEGER
The size of the array $work$. $lwork \geq n$.

Output Parameters

x1 The top part of the projected vector.
x2 The bottom part of the projected vector.
info INTEGER.
= 0: successful exit

< 0: if $info = -i$, the i -th argument has an illegal value.

See Also

[?orcsd/?uncsd](#)

[?orcsd2by1/?uncsd2by1](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

[?orbdb1/?unbdb1](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb2/?unbdb2](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb3/?unbdb3](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb4/?unbdb4](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.

[?orbdb6/?unbdb6](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.
[xerbla](#)

?orbdb6/?unbdb6

Orthogonalizes a column vector with respect to the orthonormal basis matrix.

Syntax

FORTRAN 77:

```
call sorbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call dorbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call cunbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
call zunbdb6( m1, m2, n, x1, incx1, x2, incx2, q1, ldq1, q2, ldq2, work, lwork, info )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The ?orbdb6/?unbdb6 routines orthogonalize the column vector

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

with respect to the columns of

$$q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}$$

The columns of Q must be orthonormal.

If the projection is zero according to Kahan's "twice is enough" criterion, then the zero vector is returned.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>m1</i>	INTEGER	
		The dimension of <i>x1</i> and the number of rows in <i>q1</i> . $0 \leq m1$.
<i>m2</i>	INTEGER	
		The dimension of <i>x2</i> and the number of rows in <i>q2</i> . $0 \leq m2$.
<i>n</i>	INTEGER	
		The number of columns in <i>q1</i> and <i>q2</i> . $0 \leq n$.
<i>x1</i>	REAL for sordb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5 COMPLEX*16 for zundb5	
		Array of size <i>m1</i> .
		The top part of the vector to be orthogonalized.
<i>incx1</i>	INTEGER	
		Increment for entries of <i>x1</i> .
<i>x2</i>	REAL for sordb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5 COMPLEX*16 for zundb5	
		Array of size <i>m2</i> .
		The bottom part of the vector to be orthogonalized.
<i>incx2</i>	INTEGER	
		Increment for entries of <i>x2</i> .
<i>q1</i>	REAL for sordb5 DOUBLE PRECISION for dordb5 COMPLEX for cundb5 COMPLEX*16 for zundb5	
		Array of size (<i>ldq1</i> , <i>n</i>).
		The top part of the orthonormal basis matrix.
<i>ldq1</i>	INTEGER	
		The leading dimension of <i>q1</i> . $ldq1 \geq m1$.
<i>q2</i>	REAL for sordb5	

DOUBLE PRECISION **for** dordb5
 COMPLEX **for** cundb5
 COMPLEX*16 **for** zundb5
 Array of size ($ldq2, n$).
 The bottom part of the orthonormal basis matrix.

$ldq2$ INTEGER
 The leading dimension of $q2$. $ldq2 \geq m2$.

$work$ REAL **for** sordb5
 DOUBLE PRECISION **for** dordb5
 COMPLEX **for** cundb5
 COMPLEX*16 **for** zundb5
 Workspace array of size $lwork$.

$lwork$ INTEGER
 The size of the array $work$. $lwork \geq n$.

Output Parameters

$x1$ The top part of the projected vector.
 $x2$ The bottom part of the projected vector.
 $info$ INTEGER.
 = 0: successful exit
 < 0: if $info = -i$, the i -th argument has an illegal value.

See Also

[?orcsd/?uncsd](#)
[?orcsd2by1/?uncsd2by1](#) Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.
[?orbdb1/?unbdb1](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.
[?orbdb2/?unbdb2](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.
[?orbdb3/?unbdb3](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.
[?orbdb4/?unbdb4](#) Simultaneously bidiagonalizes the blocks of a tall and skinny matrix with orthonormal columns.
[?orbdb5/?unbdb5](#) Orthogonalizes a column vector with respect to the orthonormal basis matrix.
 $xerbla$

?org2l/?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by ?geqlf (unblocked algorithm).

Syntax

```
call sorg2l( m, n, k, a, lda, tau, work, info )
call dorg2l( m, n, k, a, lda, tau, work, info )
call cung2l( m, n, k, a, lda, tau, work, info )
call zung2l( m, n, k, a, lda, tau, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?org2l/?ung2l generates an m -by- n real/complex matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m :

$$Q = H(k) * \dots * H(2) * H(1) \text{ as returned by ?geqlf.}$$

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix Q . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix Q . $m \geq n \geq 0$.
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . $n \geq k \geq 0$.
<i>a</i>	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l DOUBLE COMPLEX for zung2l. Array, DIMENSION (<i>lda,n</i>). On entry, the $(n - k + i)$ -th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?geqlf in the last k columns of its array argument <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, m)$.
<i>tau</i>	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l DOUBLE COMPLEX for zung2l. Array, DIMENSION (<i>k</i>). <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?geqlf.
<i>work</i>	REAL for sorg2l DOUBLE PRECISION for dorg2l COMPLEX for cung2l

DOUBLE COMPLEX **for** zung2l.
Workspace array, DIMENSION (n).

Output Parameters

a On exit, the m -by- n matrix Q .
info INTEGER.
 = 0: successful exit
 < 0: if *info* = $-i$, the i -th argument has an illegal value

?org2r/?ung2r

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by ?geqrf (unblocked algorithm).

Syntax

```
call sorg2r( m, n, k, a, lda, tau, work, info )
call dorg2r( m, n, k, a, lda, tau, work, info )
call cung2r( m, n, k, a, lda, tau, work, info )
call zung2r( m, n, k, a, lda, tau, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?org2r/?ung2r generates an m -by- n real/complex matrix Q with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by ?geqrf.

Input Parameters

m INTEGER. The number of rows of the matrix Q . $m \geq 0$.
n INTEGER. The number of columns of the matrix Q . $m \geq n \geq 0$.
k INTEGER. The number of elementary reflectors whose product defines the matrix Q . $n \geq k \geq 0$.
a REAL **for** sorg2r
 DOUBLE PRECISION **for** dorg2r
 COMPLEX **for** cung2r
 DOUBLE COMPLEX **for** zung2r.
Array, DIMENSION (lda, n).

On entry, the i -th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?geqrf in the first k columns of its array argument a .

<i>lda</i>	INTEGER. The first DIMENSION of the array a . $lda \geq \max(1, m)$.
<i>tau</i>	REAL for sorg2r DOUBLE PRECISION for dorg2r COMPLEX for cung2r DOUBLE COMPLEX for zung2r. Array, DIMENSION (k). $tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?geqrf.
<i>work</i>	REAL for sorg2r DOUBLE PRECISION for dorg2r COMPLEX for cung2r DOUBLE COMPLEX for zung2r. Workspace array, DIMENSION (n).

Output Parameters

<i>a</i>	On exit, the m -by- n matrix Q .
<i>info</i>	INTEGER. = 0: successful exit < 0: if $info = -i$, the i -th argument has an illegal value

?orgl2/?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by ?gelqf (unblocked algorithm).

Syntax

```
call sorgl2( m, n, k, a, lda, tau, work, info )
call dorgl2( m, n, k, a, lda, tau, work, info )
call cungl2( m, n, k, a, lda, tau, work, info )
call zungl2( m, n, k, a, lda, tau, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?orgl2/?ungl2 generates a m -by- n real/complex matrix Q with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$Q = H(k) * \dots * H(2) * H(1)$ for real flavors, or $Q = (H(k))^H * \dots * (H(2))^H * (H(1))^H$ for complex flavors as returned by ?gelqf.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix Q . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix Q . $n \geq m$.
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . $m \geq k \geq 0$.
<i>a</i>	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 DOUBLE COMPLEX for zungl2. Array, DIMENSION (<i>lda, n</i>). On entry, the <i>i</i> -th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?gelqf in the first <i>k</i> rows of its array argument <i>a</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, m)$.
<i>tau</i>	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 DOUBLE COMPLEX for zungl2. Array, DIMENSION (<i>k</i>). <i>tau(i)</i> must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?gelqf.
<i>work</i>	REAL for sorgl2 DOUBLE PRECISION for dorgl2 COMPLEX for cungl2 DOUBLE COMPLEX for zungl2. Workspace array, DIMENSION (<i>m</i>).

Output Parameters

<i>a</i>	On exit, the <i>m</i> -by- <i>n</i> matrix Q .
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument has an illegal value.

?orgqr2/?ungr2

Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by ?gerqf (unblocked algorithm).

Syntax

```
call sorgqr( m, n, k, a, lda, tau, work, info )
call dorgqr( m, n, k, a, lda, tau, work, info )
call cungqr( m, n, k, a, lda, tau, work, info )
call zunqr( m, n, k, a, lda, tau, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?orgqr2/?ungr2 generates an m -by- n real matrix Q with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

$Q = H(1) * H(2) * \dots * H(k)$ for real flavors, or $Q = (H(1))^H * (H(2))^H * \dots * (H(k))^H$ for complex flavors as returned by ?gerqf.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix Q . $m \geq 0$.
<i>n</i>	INTEGER. The number of columns of the matrix Q . $n \geq m$
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . $m \geq k \geq 0$.
<i>a</i>	REAL for sorgqr DOUBLE PRECISION for dorgqr COMPLEX for cungqr DOUBLE COMPLEX for zunqr. Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the ($m - k + i$)-th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?gerqf in the last k rows of its array argument <i>a</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, m)$.
<i>tau</i>	REAL for sorgqr DOUBLE PRECISION for dorgqr COMPLEX for cungqr DOUBLE COMPLEX for zunqr.

Array, DIMENSION (k). $\tau_{au}(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?gerqf.

work

```
REAL for sorgqr
DOUBLE PRECISION for dorgqr
COMPLEX for cungqr
DOUBLE COMPLEX for zungr2.

Workspace array, DIMENSION ( $m$ ).
```

Output Parameters

<i>a</i>	On exit, the m -by- n matrix Q .
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = $-i$, the i -th argument has an illegal value

?orm2l/?unm2l

Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by ?geqlf (unblocked algorithm).

Syntax

```
call sorm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2l( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?orm2l/?unm2l overwrites the general real/complex m -by- n matrix C with

Q^*C if *side* = 'L' and *trans* = 'N', or

$Q^T C / Q^H C$ if *side* = 'L' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors), or

C^*Q if *side* = 'R' and *trans* = 'N', or

$C^T Q^T / C^* Q^H$ if *side* = 'R' and *trans* = 'T' (for real flavors) or *trans* = 'C' (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k) * \dots * H(2) * H(1)$ as returned by ?geqlf.

Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

side CHARACTER*1.
 = 'L': apply Q or Q^T / Q^H from the left
 = 'R': apply Q or Q^T / Q^H from the right

trans CHARACTER*1.
 = 'N': apply Q (no transpose)
 = 'T': apply Q^T (transpose, for real flavors)
 = 'C': apply Q^H (conjugate transpose, for complex flavors)

m INTEGER. The number of rows of the matrix C . $m \geq 0$.

n INTEGER. The number of columns of the matrix C . $n \geq 0$.

k INTEGER. The number of elementary reflectors whose product defines the matrix Q .
 If *side* = 'L', $m \geq k \geq 0$;
 if *side* = 'R', $n \geq k \geq 0$.

a REAL for sorm21
 DOUBLE PRECISION for dorm21
 COMPLEX for cunm21
 DOUBLE COMPLEX for zunm21.
 Array, DIMENSION (*lda,k*).
 The *i*-th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by ?geqlf in the last k columns of its array argument *a*. The array *a* is modified by the routine but restored on exit.

lda INTEGER. The leading dimension of the array *a*.
 If *side* = 'L', *lda* $\geq \max(1, m)$
 if *side* = 'R', *lda* $\geq \max(1, n)$.

tau REAL for sorm21
 DOUBLE PRECISION for dorm21
 COMPLEX for cunm21
 DOUBLE COMPLEX for zunm21.
 Array, DIMENSION (*k*). *tau*(*i*) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?geqlf.

c REAL for sorm21
 DOUBLE PRECISION for dorm21
 COMPLEX for cunm21
 DOUBLE COMPLEX for zunm21.

Array, DIMENSION (ldc, n).
On entry, the m -by- n matrix C .

ldc INTEGER. The leading dimension of the array C . $ldc \geq \max(1, m)$.

$work$ REAL for sorm2l
 DOUBLE PRECISION for dorm2l
 COMPLEX for cunm2l
 DOUBLE COMPLEX for zunm2l.

Workspace array, DIMENSION:
 (n) if $side = 'L'$,
 (m) if $side = 'R'$.

Output Parameters

c On exit, c is overwritten by Q^*C or $Q^T*C / Q^{H*}C$, or C^*Q , or C^*Q^T / C^*Q^H .

$info$ INTEGER.
 = 0: successful exit
 < 0: if $info = -i$, the i -th argument had an illegal value

?orm2r/?unm2r

Multiples a general matrix by the orthogonal/unitary matrix from a QR factorization determined by ?geqrf (unblocked algorithm).

Syntax

```
call sorm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunm2r( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?orm2r/?unm2r overwrites the general real/complex m -by- n matrix C with Q^*C if $side = 'L'$ and $trans = 'N'$, or $Q^T*C / Q^{H*}C$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or C^*Q if $side = 'R'$ and $trans = 'N'$, or C^*Q^T / C^*Q^H if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors). Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors $Q = H(1)*H(2)*\dots*H(k)$ as returned by ?geqrf.

Q is of order *m* if *side* = 'L' and of order *n* if *side* = 'R'.

Input Parameters

<i>side</i>	CHARACTER*1. = 'L': apply <i>Q</i> or Q^T / Q^H from the left = 'R': apply <i>Q</i> or Q^T / Q^H from the right
<i>trans</i>	CHARACTER*1. = 'N': apply <i>Q</i> (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<i>m</i>	INTEGER. The number of rows of the matrix <i>C</i> . <i>m</i> ≥ 0 .
<i>n</i>	INTEGER. The number of columns of the matrix <i>C</i> . <i>n</i> ≥ 0 .
<i>k</i>	INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> . If <i>side</i> = 'L', <i>m</i> $\geq k \geq 0$; if <i>side</i> = 'R', <i>n</i> $\geq k \geq 0$.
<i>a</i>	REAL for sorm2r DOUBLE PRECISION for dorm2r COMPLEX for cunm2r DOUBLE COMPLEX for zunm2r. Array, DIMENSION (<i>lda,k</i>). The <i>i</i> -th column must contain the vector which defines the elementary reflector $H(i)$, for <i>i</i> = 1, 2, ..., <i>k</i> , as returned by ?geqrf in the first <i>k</i> columns of its array argument <i>a</i> . The array <i>a</i> is modified by the routine but restored on exit.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . If <i>side</i> = 'L', <i>lda</i> $\geq \max(1, m)$; if <i>side</i> = 'R', <i>lda</i> $\geq \max(1, n)$.
<i>tau</i>	REAL for sorm2r DOUBLE PRECISION for dorm2r COMPLEX for cunm2r DOUBLE COMPLEX for zunm2r. Array, DIMENSION (<i>k</i>). <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?geqrf.
<i>c</i>	REAL for sorm2r DOUBLE PRECISION for dorm2r

COMPLEX for `cunm2r`
 DOUBLE COMPLEX for `zunm2r`.
Array, DIMENSION (ldc, n).
 On entry, the m -by- n matrix C .

ldc INTEGER. The leading dimension of the array c . $ldc \geq \max(1, m)$.

work REAL for `sorm2r`
 DOUBLE PRECISION for `dorm2r`
 COMPLEX for `cunm2r`
 DOUBLE COMPLEX for `zunm2r`.
Workspace array, DIMENSION
 (n) if $\text{side} = 'L'$,
 (m) if $\text{side} = 'R'$.

Output Parameters

c On exit, c is overwritten by Q^*C or $Q^T*C / Q^{H*}C$, or C^*Q , or C^*Q^T / C^*Q^H .

info INTEGER.
 = 0: successful exit
 < 0: if $\text{info} = -i$, the i -th argument had an illegal value

?orml2/?unml2

Multiples a general matrix by the orthogonal/unitary matrix from a LQ factorization determined by ?gelqf (unblocked algorithm).

Syntax

```
call sorml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dorml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunml2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine `?orml2/?unml2` overwrites the general real/complex m -by- n matrix C with

Q^*C if $\text{side} = 'L'$ and $\text{trans} = 'N'$, or
 $Q^T*C / Q^{H*}C$ if $\text{side} = 'L'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors), or
 C^*Q if $\text{side} = 'R'$ and $\text{trans} = 'N'$, or
 C^*Q^T / C^*Q^H if $\text{side} = 'R'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k) * \dots * H(2) * H(1)$ for real flavors, or $Q = (H(k))^H * \dots * (H(2))^H * (H(1))^H$ for complex flavors as returned by `?gelqf`.

Q is of order m if `side` = 'L' and of order n if `side` = 'R'.

Input Parameters

<code>side</code>	CHARACTER*1. = 'L': apply Q or Q^T / Q^H from the left = 'R': apply Q or Q^T / Q^H from the right
<code>trans</code>	CHARACTER*1. = 'N': apply Q (no transpose) = 'T': apply Q^T (transpose, for real flavors) = 'C': apply Q^H (conjugate transpose, for complex flavors)
<code>m</code>	INTEGER. The number of rows of the matrix C . $m \geq 0$.
<code>n</code>	INTEGER. The number of columns of the matrix C . $n \geq 0$.
<code>k</code>	INTEGER. The number of elementary reflectors whose product defines the matrix Q . If <code>side</code> = 'L', $m \geq k \geq 0$; if <code>side</code> = 'R', $n \geq k \geq 0$.
<code>a</code>	REAL for <code>sorml2</code> DOUBLE PRECISION for <code>dorml2</code> COMPLEX for <code>cunml2</code> DOUBLE COMPLEX for <code>zunml2</code> . Array, DIMENSION (<code>lda, m</code>) if <code>side</code> = 'L', (<code>lda, n</code>) if <code>side</code> = 'R' The i -th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by <code>?gelqf</code> in the first k rows of its array argument <code>a</code> . The array <code>a</code> is modified by the routine but restored on exit.
<code>lda</code>	INTEGER. The leading dimension of the array <code>a</code> . $lda \geq \max(1, k)$.
<code>tau</code>	REAL for <code>sorml2</code> DOUBLE PRECISION for <code>dorml2</code> COMPLEX for <code>cunml2</code> DOUBLE COMPLEX for <code>zunml2</code> . Array, DIMENSION (k).

$\tau_{au}(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by `?gelqf`.

c REAL for `sorml2`
 DOUBLE PRECISION for `dorml2`
 COMPLEX for `cunml2`
 DOUBLE COMPLEX for `zunml2`.
 Array, DIMENSION (*ldc*, *n*) On entry, the *m*-by-*n* matrix *C*.
ldc INTEGER. The leading dimension of the array *c*. *ldc* $\geq \max(1, m)$.
work REAL for `sorml2`
 DOUBLE PRECISION for `dorml2`
 COMPLEX for `cunml2`
 DOUBLE COMPLEX for `zunml2`.
 Workspace array, DIMENSION
 (*n*) if *side* = 'L',
 (*m*) if *side* = 'R'

Output Parameters

c On exit, *c* is overwritten by Q^*C or Q^T*C / $Q^{H*}C$, or C^*Q , or C^*Q^T / C^*Q^H .
info INTEGER.
 = 0: successful exit
 < 0: if *info* = -*i*, the *i*-th argument had an illegal value

?ormr2/?unmr2

Multiples a general matrix by the orthogonal/unitary matrix from a RQ factorization determined by `?gerqf` (unblocked algorithm).

Syntax

```
call sormr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call dormr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call cunmr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
call zunmr2( side, trans, m, n, k, a, lda, tau, c, ldc, work, info )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine `?ormr2/?unmr2` overwrites the general real/complex *m*-by-*n* matrix *C* with Q^*C if *side* = 'L' and *trans* = 'N', or

$Q^T C / Q^H C$ if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

$C^* Q$ if $side = 'R'$ and $trans = 'N'$, or

$C^* Q^T / C^* Q^H$ if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(1)^* H(2)^* \dots^* H(k)$ for real flavors, or $Q = (H(1))^H * (H(2))^H * \dots * (H(k))^H$ as returned by `?gerqf`.

Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$

CHARACTER*1.

= 'L': apply Q or Q^T / Q^H from the left

= 'R': apply Q or Q^T / Q^H from the right

$trans$

CHARACTER*1.

= 'N': apply Q (no transpose)

= 'T': apply Q^T (transpose, for real flavors)

= 'C': apply Q^H (conjugate transpose, for complex flavors)

m

INTEGER. The number of rows of the matrix C . $m \geq 0$.

n

INTEGER. The number of columns of the matrix C . $n \geq 0$.

k

INTEGER. The number of elementary reflectors whose product defines the matrix Q .

If $side = 'L'$, $m \geq k \geq 0$;

if $side = 'R'$, $n \geq k \geq 0$.

a

REAL for `sormr2`

DOUBLE PRECISION for `dormr2`

COMPLEX for `cunmr2`

DOUBLE COMPLEX for `zunmr2`.

Array, DIMENSION

(lda, m) if $side = 'L'$,

(lda, n) if $side = 'R'$

The i -th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by `?gerqf` in the last k rows of its array argument a . The array a is modified by the routine but restored on exit.

lda

INTEGER.

The leading dimension of the array a . $lda \geq \max(1, k)$.

tau

REAL for `sormr2`

DOUBLE PRECISION for `dormr2`

COMPLEX for `cunmr2`

DOUBLE COMPLEX for zunmr2.
Array, DIMENSION (k).
tau(i) must contain the scalar factor of the elementary reflector $H(i)$, as returned by ?gerqf.

c
REAL for sormr2
DOUBLE PRECISION for dormr2
COMPLEX for cunmr2
DOUBLE COMPLEX for zunmr2.
Array, DIMENSION (ldc, n).
On entry, the m -by- n matrix C .

ldc
INTEGER. The leading dimension of the array c . $ldc \geq \max(1, m)$.

work
REAL for sormr2
DOUBLE PRECISION for dormr2
COMPLEX for cunmr2
DOUBLE COMPLEX for zunmr2.
Workspace array, DIMENSION
(n) if *side* = 'L',
(m) if *side* = 'R'

Output Parameters

c
On exit, c is overwritten by Q^*C or Q^H*C / Q^H*Q , or C^*Q , or C^*Q^T / C^*Q^H .

info
INTEGER.
= 0: successful exit
< 0: if *info* = $-i$, the i -th argument had an illegal value

?ormr3/?unmr3

Multiples a general matrix by the orthogonal/unitary matrix from a RZ factorization determined by ?tzrzf (unblocked algorithm).

Syntax

```
call sormr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call dormr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call cunmr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
call zunmr3( side, trans, m, n, k, l, a, lda, tau, c, ldc, work, info )
```

Include Files

- Fortran: mkl.fi

- C: mkl.h

Description

The routine ?ormr3/?unmr3 overwrites the general real/complex m -by- n matrix C with

Q^*C if $side = 'L'$ and $trans = 'N'$, or

Q^T*C / Q^H*C if $side = 'L'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors), or

C^*Q if $side = 'R'$ and $trans = 'N'$, or

C^*Q^T / C^*Q^H if $side = 'R'$ and $trans = 'T'$ (for real flavors) or $trans = 'C'$ (for complex flavors).

Here Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(1)*H(2)*\dots*H(k)$ as returned by ?tzrzf.

Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

side

CHARACTER*1.

= 'L': apply Q or Q^T / Q^H from the left

= 'R': apply Q or Q^T / Q^H from the right

trans

CHARACTER*1.

= 'N': apply Q (no transpose)

= 'T': apply Q^T (transpose, for real flavors)

= 'C': apply Q^H (conjugate transpose, for complex flavors)

m

INTEGER. The number of rows of the matrix C . $m \geq 0$.

n

INTEGER. The number of columns of the matrix C . $n \geq 0$.

k

INTEGER. The number of elementary reflectors whose product defines the matrix Q .

If $side = 'L'$, $m \geq k \geq 0$;

If $side = 'R'$, $n \geq k \geq 0$.

l

INTEGER. The number of columns of the matrix A containing the meaningful part of the Householder reflectors.

If $side = 'L'$, $m \geq l \geq 0$,

If $side = 'R'$, $n \geq l \geq 0$.

a

REAL for sormr3

DOUBLE PRECISION for dormr3

COMPLEX for cunmr3

DOUBLE COMPLEX for zunmr3.

Array, DIMENSION

(*lda*, *m*) if $side = 'L'$,

(*lda*, *n*) if $side = 'R'$

The i -th row must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \dots, k$, as returned by `?tzrzf` in the last k rows of its array argument a . The array a is modified by the routine but restored on exit.

lda

INTEGER.

The leading dimension of the array a . $lda \geq \max(1, k)$.*tau*REAL for `sormr3`DOUBLE PRECISION for `dormr3`COMPLEX for `cunmr3`DOUBLE COMPLEX for `zunmr3`.Array, DIMENSION (k). $tau(i)$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by `?tzrzf`.*c*REAL for `sormr3`DOUBLE PRECISION for `dormr3`COMPLEX for `cunmr3`DOUBLE COMPLEX for `zunmr3`.Array, DIMENSION (ldc, n).On entry, the m -by- n matrix C .*ldc*INTEGER. The leading dimension of the array c . $ldc \geq \max(1, m)$.*work*REAL for `sormr3`DOUBLE PRECISION for `dormr3`COMPLEX for `cunmr3`DOUBLE COMPLEX for `zunmr3`.

Workspace array, DIMENSION

 (n) if $side = 'L'$, (m) if $side = 'R'$.

Output Parameters

*c*On exit, c is overwritten by Q^*C or $Q^T*C / Q^{H*}C$, or C^*Q , or C^*Q^T / C^*Q^H .*info*

INTEGER.

= 0: successful exit

< 0: if $info = -i$, the i -th argument had an illegal value

?pbtf2

*Computes the Cholesky factorization of a symmetric/
Hermitian positive-definite band matrix (unblocked
algorithm).*

Syntax

```
call spbtf2( uplo, n, kd, ab, ldab, info )
call dpbtf2( uplo, n, kd, ab, ldab, info )
call cpbtf2( uplo, n, kd, ab, ldab, info )
call zpbtf2( uplo, n, kd, ab, ldab, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite band matrix A .

The factorization has the form

$A = U^T * U$ for real flavors, $A = U^H * U$ for complex flavors if $uplo = 'U'$, or

$A = L * L^T$ for real flavors, $A = L * L^H$ for complex flavors if $uplo = 'L'$,

where U is an upper triangular matrix, and L is lower triangular. This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

<i>uplo</i>	CHARACTER*1.
	Specifies whether the upper or lower triangular part of the symmetric/ Hermitian matrix A is stored:
	= 'U': upper triangular
	= 'L': lower triangular
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>kd</i>	INTEGER. The number of super-diagonals of the matrix A if $uplo = 'U'$, or the number of sub-diagonals if $uplo = 'L'$. $kd \geq 0$.
<i>ab</i>	REAL for spbtf2 DOUBLE PRECISION for dpbtf2 COMPLEX for cpbtf2 DOUBLE COMPLEX for zpbtf2. Array, DIMENSION (<i>ldab</i> , <i>n</i>).

On entry, the upper or lower triangle of the symmetric/ Hermitian band matrix A , stored in the first $kd+1$ rows of the array. The j -th column of A is stored in the j -th column of the array ab as follows:

```
if uplo = 'U', ab(kd+1+i-j, j) = A(i, j for  $\max(1, j-kd) \leq i \leq j$ );  
if uplo = 'L', ab(1+i-j, j) = A(i, j for  $j \leq i \leq \min(n, j+kd)$ ).
```

ldab

INTEGER. The leading dimension of the array ab . $ldab \geq kd+1$.

Output Parameters

ab

On exit, If *info* = 0, the triangular factor U or L from the Cholesky factorization $A=U^T \star U$ ($A=U^H \star U$), or $A= L \star L^T$ ($A= L \star L^H$) of the band matrix A , in the same storage format as A .

info

INTEGER.

= 0: successful exit

< 0: if *info* = $-k$, the k -th argument had an illegal value

> 0: if *info* = k , the leading minor of order k is not positive definite, and the factorization could not be completed.

?potf2

Computes the Cholesky factorization of a symmetric/ Hermitian positive-definite matrix (unblocked algorithm).

Syntax

```
call spotf2( uplo, n, a, lda, info )  
call dpotf2( uplo, n, a, lda, info )  
call cpotf2( uplo, n, a, lda, info )  
call zpotf2( uplo, n, a, lda, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?potf2 computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite matrix A . The factorization has the form

$A = U^T \star U$ for real flavors, $A = U^H \star U$ for complex flavors if *uplo* = 'U', or

$A = L \star L^T$ for real flavors, $A = L \star L^H$ for complex flavors if *uplo* = 'L',

where U is an upper triangular matrix, and L is lower triangular.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#)

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix A is stored.

- = 'U': upper triangular
- = 'L': lower triangular

n INTEGER. The order of the matrix A . $n \geq 0$.

a REAL for `sptf2`
 DOUBLE PRECISION or `dpotf2`
 COMPLEX for `cpotf2`
 DOUBLE COMPLEX for `zpotf2`.

Array, DIMENSION (*lda*, *n*).

On entry, the symmetric/Hermitian matrix A .

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix A , and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix A , and the strictly upper triangular part of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*.

lda $\geq \max(1, n)$.

Output Parameters

a On exit, If *info* = 0, the factor U or L from the Cholesky factorization $A = U^T * U$ ($A = U^H * U$), or $A = L * L^T$ ($A = L * L^H$).

info INTEGER.
 = 0: successful exit
 < 0: if *info* = $-k$, the *k*-th argument had an illegal value
 > 0: if *info* = *k*, the leading minor of order *k* is not positive definite, and the factorization could not be completed.

?ptts2

Solves a tridiagonal system of the form $A * X = B$ using the $L * D * L^H / L * D * L^H$ factorization computed by ?pttrf.

Syntax

```
call sptts2( n, nrhs, d, e, b, ldb )
call dptts2( n, nrhs, d, e, b, ldb )
call cptts2( iuplo, n, nrhs, d, e, b, ldb )
call zptts2( iuplo, n, nrhs, d, e, b, ldb )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?ptts2 solves a tridiagonal system of the form

$$A^*X = B$$

Real flavors sptts2/dptts2 use the $L^*D^*L^T$ factorization of A computed by [spttrf/dpttrf](#), and complex flavors cptts2/zptts2 use the U^H*D^*U or $L^*D^*L^H$ factorization of A computed by [cpttrf/zpttrf](#).

D is a diagonal matrix specified in the vector d , U (or L) is a unit bidiagonal matrix whose superdiagonal (subdiagonal) is specified in the vector e , and X and B are n -by- $nrhs$ matrices.

Input Parameters

<i>iuplo</i>	INTEGER. Used with complex flavors only. Specifies the form of the factorization, and whether the vector e is the superdiagonal of the upper bidiagonal factor U or the subdiagonal of the lower bidiagonal factor L . = 1: $A = U^H*D^*U$, e is the superdiagonal of U ; = 0: $A = L^*D^*L^H$, e is the subdiagonal of L
<i>n</i>	INTEGER. The order of the tridiagonal matrix A . $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right hand sides, that is, the number of columns of the matrix B . $nrhs \geq 0$.
<i>d</i>	REAL for sptts2/cptts2 DOUBLE PRECISION for dptts2/zptts2. Array, DIMENSION (n). The n diagonal elements of the diagonal matrix D from the factorization of A .
<i>e</i>	REAL for sptts2 DOUBLE PRECISION for dptts2 COMPLEX for cptts2 DOUBLE COMPLEX for zptts2. Array, DIMENSION ($n-1$). Contains the $(n-1)$ subdiagonal elements of the unit bidiagonal factor L from the $L^*D^*L^T$ (for real flavors) or $L^*D^*L^H$ (for complex flavors when <i>iuplo</i> = 0) factorization of A . For complex flavors when <i>iuplo</i> = 1, e contains the $(n-1)$ superdiagonal elements of the unit bidiagonal factor U from the factorization $A = U^H*D^*U$.
<i>B</i>	REAL for sptts2/cptts2 DOUBLE PRECISION for dptts2/zptts2. Array, DIMENSION ($ldb, nrhs$).

On entry, the right hand side vectors B for the system of linear equations.

ldb INTEGER. The leading dimension of the array B . $ldb \geq \max(1, n)$.

Output Parameters

b On exit, the solution vectors, X .

?rscl

Multiples a vector by the reciprocal of a real scalar.

Syntax

```
call srscl( n, sa, sx, incx )
call drscl( n, sa, sx, incx )
call csrsc1( n, sa, sx, incx )
call zdrscl( n, sa, sx, incx )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?rscl multiplies an n -element real/complex vector x by the real scalar $1/a$. This is done without overflow or underflow as long as the final result x/a does not overflow or underflow.

Input Parameters

n INTEGER. The number of components of the vector x .

sa REAL for srscl/csrsc1
DOUBLE PRECISION for drscl/zdrscl.

The scalar a which is used to divide each component of the vector x . sa must be ≥ 0 , or the subroutine will divide by zero.

sx REAL for srscl
DOUBLE PRECISION for drscl
COMPLEX for csrsc1
DOUBLE COMPLEX for zdrscl.
Array, DIMENSION (1+(n-1)* |incx|).

The n -element vector x .

$incx$ INTEGER. The increment between successive values of the vector sx .
If $incx > 0$, $sx(1)=x(1)$, and $sx(1+(i-1)*incx)=x(i)$, $1 < i \leq n$.

Output Parameters

sx On exit, the result x/a .

?syswapr

Applies an elementary permutation on the rows and columns of a symmetric matrix.

Syntax

FORTRAN 77:

```
call ssyswapr( uplo, n, a, i1, i2 )
call dsyswapr( uplo, n, a, i1, i2 )
call csyswapr( uplo, n, a, i1, i2 )
call zsyswapr( uplo, n, a, i1, i2 )
```

Fortran 95:

```
call syswapr( a, i1, i2[, uplo] )
```

C:

```
lapack_int LAPACKE_<?>syswapr (int matrix_layout, char uplo, lapack_int n, <datatype> * a, lapack_int i1, lapack_int i2);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine applies an elementary permutation on the rows and columns of a symmetric matrix.

Input Parameters

The data types are given for the Fortran interface.

uplo CHARACTER*1. Must be 'U' or 'L'.

Indicates how the input matrix A has been factored:

If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = U \cdot D \cdot U^T$.

If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = L \cdot D \cdot L^T$.

n INTEGER. The order of matrix A ; $n \geq 0$.

nrhs INTEGER. The number of right-hand sides; $nrhs \geq 0$.

a REAL for ssyswapr

DOUBLE PRECISION for dsyswapr

COMPLEX for csyswapr

DOUBLE COMPLEX for zsyswapr

Array of dimension (lda, n).

The array a contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?sytrf.

$i1$ INTEGER. Index of the first row to swap.

$i2$ INTEGER. Index of the second row to swap.

Output Parameters

a If $info = 0$, the symmetric inverse of the original matrix.

If $info = 'U'$, the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced.

If $info = 'L'$, the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine syswapr interface are as follows:

a Holds the matrix A of size (n, n).

$i1$ Holds the index for swap.

$i2$ Holds the index for swap.

$uplo$ Indicates how the matrix A has been factored. Must be ' U ' or ' L '.

See Also

?sytrf

?heswapr

Applies an elementary permutation on the rows and columns of a Hermitian matrix.

Syntax

FORTRAN 77:

```
call cheswapr( uplo, n, a, i1, i2 )
call zheswapr( uplo, n, a, i1, i2 )
```

Fortran 95:

```
call heswapr( a, i1, i2 [,uplo] )
```

C:

```
lapack_int LAPACKE_<?>heswapr (int matrix_layout, char uplo, lapack_int n, <datatype> * a, lapack_int i1, lapack_int i2);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The routine applies an elementary permutation on the rows and columns of a Hermitian matrix.

Input Parameters

The data types are given for the Fortran interface.

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'.
	Indicates how the input matrix A has been factored:
	If <i>uplo</i> = 'U', the array <i>a</i> stores the upper triangular factor U of the factorization $A = U^*D^*U^H$.
	If <i>uplo</i> = 'L', the array <i>a</i> stores the lower triangular factor L of the factorization $A = L^*D^*L^H$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a</i>	COMPLEX for cheswapr DOUBLE COMPLEX for zheswapr Array of dimension (<i>lda</i> , <i>n</i>). The array <i>a</i> contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?hetrf.
<i>i1</i>	INTEGER. Index of the first row to swap.
<i>i2</i>	INTEGER. Index of the second row to swap.

Output Parameters

<i>a</i>	If <i>info</i> = 0, the inverse of the original matrix. If <i>info</i> = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced. If <i>info</i> = 'L', the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.
----------	---

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine `heswapr` interface are as follows:

<i>a</i>	Holds the matrix A of size (<i>n</i> , <i>n</i>).
----------	---

<i>i1</i>	Holds the index for swap.
<i>i2</i>	Holds the index for swap.
<i>uplo</i>	Must be 'U' or 'L'.

See Also[?hetrf](#)[?syswapr1](#)**?syswapr1**

Applies an elementary permutation on the rows and columns of a symmetric matrix.

Syntax**FORTRAN 77:**

```
call ssyswapr1( uplo, n, a, i1, i2 )
call dsyswapr1( uplo, n, a, i1, i2 )
call csyswapr1( uplo, n, a, i1, i2 )
call zsyswapr1( uplo, n, a, i1, i2 )
```

Fortran 95:

```
call syswapr1( a,i1,i2[,uplo] )
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90

Description

The routine applies an elementary permutation on the rows and columns of a symmetric matrix.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Indicates how the input matrix A has been factored: If $uplo = 'U'$, the array a stores the upper triangular factor U of the factorization $A = U \cdot D \cdot U^T$. If $uplo = 'L'$, the array a stores the lower triangular factor L of the factorization $A = L \cdot D \cdot L^T$.
<i>n</i>	INTEGER. The order of matrix A ; $n \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; $nrhs \geq 0$.
<i>a</i>	REAL for ssyswapr1 DOUBLE PRECISION for dsyswapr1 COMPLEX for csyswapr1 DOUBLE COMPLEX for zsyswapr1

Array of dimension (lda, n) .

The array a contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?sytrf .

lda INTEGER. The leading dimension of the array a . $\text{lda} \geq \max(1, n)$.

$i1$ INTEGER. Index of the first row to swap.

$i2$ INTEGER. Index of the second row to swap.

Output Parameters

a If $\text{info} = 0$, the symmetric inverse of the original matrix.

If $\text{info} = 'U'$, the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced.

If $\text{info} = 'L'$, the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

Fortran 95 Interface Notes

Routines in Fortran 95 interface have fewer arguments in the calling sequence than their FORTRAN 77 counterparts. For general conventions applied to skip redundant or reconstructible arguments, see [Fortran 95 Interface Conventions](#).

Specific details for the routine syswapr1 interface are as follows:

a Holds the matrix A of size (n, n) .

$i1$ Holds the index for swap.

$i2$ Holds the index for swap.

uplo Indicates how the matrix A has been factored. Must be ' U ' or ' L '.

See Also

[?sytrf](#)

?sygs2/?hegs2

Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from ?potrf (unblocked algorithm).

Syntax

```
call ssygs2( itype, uplo, n, a, lda, b, ldb, info )
call dsygs2( itype, uplo, n, a, lda, b, ldb, info )
call chegs2( itype, uplo, n, a, lda, b, ldb, info )
call zhegs2( itype, uplo, n, a, lda, b, ldb, info )
```

Include Files

- Fortran: `mkl.fi`

- C: mkl.h

Description

The routine ?sygs2/?hegs2 reduces a real symmetric-definite or a complex Hermitian positive-definite generalized eigenproblem to standard form.

If *itype* = 1, the problem is

$$A^*x = \lambda * B^*x$$

and *A* is overwritten by $\text{inv}(U^H) * A * \text{inv}(U)$ or $\text{inv}(L) * A * \text{inv}(L^H)$ for complex flavors and by $\text{inv}(U^T) * A * \text{inv}(U)$ or $\text{inv}(L) * A * \text{inv}(L^T)$ for real flavors.

If *itype* = 2 or 3, the problem is

$$A^*B^*x = \lambda * x, \text{ or } B^*A^*x = \lambda * x,$$

and *A* is overwritten by $U^*A^*U^H$ or $L^H * A * L$ for complex flavors and by $U^*A^*U^T$ or $L^T * A * L$ for real flavors. Here U^T and L^T are the transpose while U^H and L^H are conjugate transpose of *U* and *L*.

B must be previously factorized by ?potrf as follows:

- $U^H * U$ or $L * L^H$ for complex flavors
- $U^T * U$ or $L^T * L$ for real flavors

Input Parameters

itype

INTEGER.

For complex flavors:

- = 1: compute $\text{inv}(U^H) * A * \text{inv}(U)$ or $\text{inv}(L) * A * \text{inv}(L^H)$;
- = 2 or 3: compute $U^*A^*U^H$ or $L^H * A * L$.

For real flavors:

- = 1: compute $\text{inv}(U^T) * A * \text{inv}(U)$ or $\text{inv}(L) * A * \text{inv}(L^T)$;
- = 2 or 3: compute $U^*A^*U^T$ or $L^T * A * L$.

uplo

CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix *A* is stored, and how *B* has been factorized.

- = 'U': upper triangular
- = 'L': lower triangular

n

INTEGER. The order of the matrices *A* and *B*. $n \geq 0$.

a

REAL for ssygs2

DOUBLE PRECISION for dsygs2

COMPLEX for chegs2

DOUBLE COMPLEX for zhegs2.

Array, DIMENSION (*lda*, *n*).

On entry, the symmetric/Hermitian matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

lda

INTEGER.

The leading dimension of the array a . $lda \geq \max(1, n)$.*b*

REAL for ssygs2

DOUBLE PRECISION for dsygs2

COMPLEX for chegs2

DOUBLE COMPLEX for zhegs2.

Array, DIMENSION (ldb, n).The triangular factor from the Cholesky factorization of B as returned by ?potrf.*ldb*INTEGER. The leading dimension of the array b . $ldb \geq \max(1, n)$.

Output Parameters

*a*On exit, If $info = 0$, the transformed matrix, stored in the same format as A .*info*

INTEGER.

= 0: successful exit.

< 0: if $info = -i$, the i -th argument had an illegal value.

?sytd2/?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation(unblocked algorithm).

Syntax

```
call ssytd2( uplo, n, a, lda, d, e, tau, info )
call dsytd2( uplo, n, a, lda, d, e, tau, info )
call chetd2( uplo, n, a, lda, d, e, tau, info )
call zhetd2( uplo, n, a, lda, d, e, tau, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?sytd2/?hetd2 reduces a real symmetric/complex Hermitian matrix A to real symmetric tridiagonal form T by an orthogonal/unitary similarity transformation: $Q^T A Q = T$ ($Q^H A Q = T$).

Input Parameters

uplo CHARACTER*1.
 Specifies whether the upper or lower triangular part of the symmetric/
 Hermitian matrix *A* is stored:
 = 'U': upper triangular
 = 'L': lower triangular

n INTEGER. The order of the matrix *A*. $n \geq 0$.

a REAL for ssytd2
 DOUBLE PRECISION for dsytd2
 COMPLEX for chetd2
 DOUBLE COMPLEX for zhetd2.
 Array, DIMENSION (*lda*, *n*).
 On entry, the symmetric/Hermitian matrix *A*.
 If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the
 upper triangular part of the matrix *A*, and the strictly lower triangular part
 of *a* is not referenced.
 If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the
 lower triangular part of the matrix *A*, and the strictly upper triangular part
 of *a* is not referenced.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a On exit, if *uplo* = 'U', the diagonal and first superdiagonal of *a* are
 overwritten by the corresponding elements of the tridiagonal matrix *T*, and
 the elements above the first superdiagonal, with the array *tau*, represent
 the orthogonal/unitary matrix *Q* as a product of elementary reflectors;
 if *uplo* = 'L', the diagonal and first subdiagonal of *a* are overwritten by
 the corresponding elements of the tridiagonal matrix *T*, and the elements
 below the first subdiagonal, with the array *tau*, represent the orthogonal/
 unitary matrix *Q* as a product of elementary reflectors.

d REAL for ssytd2/chetd2
 DOUBLE PRECISION for dsytd2/zhetd2.
 Array, DIMENSION (*n*).
 The diagonal elements of the tridiagonal matrix *T*:

$$d(i) = a(i, i).$$

e REAL for ssytd2/chetd2
 DOUBLE PRECISION for dsytd2/zhetd2.
 Array, DIMENSION (*n*-1).
 The off-diagonal elements of the tridiagonal matrix *T*:

$e(i) = a(i,i+1)$ if $uplo = 'U'$,
 $e(i) = a(i+1,i)$ if $uplo = 'L'$.

tau REAL for ssytd2
 DOUBLE PRECISION for dsytd2
 COMPLEX for chetd2
 DOUBLE COMPLEX for zhetd2.
 Array, DIMENSION (n).

The first $n-1$ elements contain scalar factors of the elementary reflectors. *tau(n)* is used as workspace.

info INTEGER.
 = 0: successful exit
 < 0: if *info* = $-i$, the i -th argument had an illegal value.

?sytf2

Computes the factorization of a real/complex symmetric indefinite matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call ssytf2( uplo, n, a, lda, ipiv, info )
call dsytf2( uplo, n, a, lda, ipiv, info )
call csytf2( uplo, n, a, lda, ipiv, info )
call zsytf2( uplo, n, a, lda, ipiv, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?sytf2 computes the factorization of a real/complex symmetric matrix A using the Bunch-Kaufman diagonal pivoting method:

$$A = U^* D^* U^T, \text{ or } A = L^* D^* L^T,$$

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

uplo CHARACTER*1.
 Specifies whether the upper or lower triangular part of the symmetric matrix A is stored
 = 'U': upper triangular

= 'L': lower triangular

n INTEGER. The order of the matrix *A*. $n \geq 0$.

a REAL for ssytf2

DOUBLE PRECISION for dsytf2

COMPLEX for csytf2

DOUBLE COMPLEX for zsytf2.

Array, DIMENSION (*lda*, *n*).

On entry, the symmetric matrix *A*.

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of *a* contains the upper triangular part of the matrix *A*, and the strictly lower triangular part of *a* is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of *a* contains the lower triangular part of the matrix *A*, and the strictly upper triangular part of *a* is not referenced.

lda INTEGER.

The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a On exit, the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L*.

ipiv INTEGER.

Array, DIMENSION (*n*).

Details of the interchanges and the block structure of *D*

If *ipiv*(*k*) > 0, then rows and columns *k* and *ipiv*(*k*) are interchanged and *D*(*k,k*) is a 1-by-1 diagonal block.

If *uplo* = 'U' and *ipiv*(*k*) = *ipiv*(*k-1*) < 0, then rows and columns *k-1* and -*ipiv*(*k*) are interchanged and *D*(*k-1:k, k-1:k*) is a 2-by-2 diagonal block.

If *uplo* = 'L' and *ipiv*(*k*) = *ipiv*(*k+1*) < 0, then rows and columns *k+1* and -*ipiv*(*k*) were interchanged and *D*(*k:k+1, k:k+1*) is a 2-by-2 diagonal block.

info INTEGER.

= 0: successful exit

< 0: if *info* = -*k*, the *k*-th argument has an illegal value

> 0: if *info* = *k*, *D*(*k,k*) is exactly zero. The factorization are completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?sytf2_rook

Computes the factorization of a real/complex symmetric indefinite matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).

Syntax

```
call ssytf2_rook( uplo, n, a, lda, ipiv, info )
call dsytf2_rook( uplo, n, a, lda, ipiv, info )
call csytf2_rook( uplo, n, a, lda, ipiv, info )
call zsytf2_rook( uplo, n, a, lda, ipiv, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?sytf2_rook computes the factorization of a real/complex symmetric matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method:

$$A = U^* D^* U^T, \text{ or } A = L^* D^* L^T,$$

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric matrix A is stored = 'U': upper triangular = 'L': lower triangular
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>a</i>	REAL for ssytf2_rook DOUBLE PRECISION for dsytf2_rook COMPLEX for csytf2_rook DOUBLE COMPLEX for zsytf2_rook. Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the symmetric matrix A . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of the matrix A , and the strictly lower triangular part of <i>a</i> is not referenced.

If $\text{uplo} = \text{'L'}$, the leading n -by- n lower triangular part of a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

lda INTEGER.

The leading dimension of the array a . $\text{lda} \geq \max(1, n)$.

Output Parameters

a On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L .

ipiv INTEGER.

Array, DIMENSION (n).

Details of the interchanges and the block structure of D

If $\text{ipiv}(k) > 0$, then rows and columns k and $\text{ipiv}(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $\text{uplo} = \text{'U'}$ and $\text{ipiv}(k) < 0$ and $\text{ipiv}(k - 1) < 0$, then rows and columns k and $-\text{ipiv}(k)$ were interchanged, rows and columns $k - 1$ and $-\text{ipiv}(k - 1)$ were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.

If $\text{uplo} = \text{'L'}$ and $\text{ipiv}(k) < 0$ and $\text{ipiv}(k + 1) < 0$, then rows and columns k and $-\text{ipiv}(k)$ were interchanged, rows and columns $k + 1$ and $-\text{ipiv}(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

info INTEGER.

= 0: successful exit

< 0: if $\text{info} = -k$, the k -th argument has an illegal value

> 0: if $\text{info} = k$, $D(k,k)$ is exactly zero. The factorization are completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?hetf2

Computes the factorization of a complex Hermitian matrix, using the diagonal pivoting method (unblocked algorithm).

Syntax

```
call chetf2( uplo, n, a, lda, ipiv, info )
call zhetf2( uplo, n, a, lda, ipiv, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method:

$A = U^* D^* U^H$ or $A = L^* D^* L^H$

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, U^H is the conjugate transpose of U , and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>A</i>	COMPLEX for <code>chetf2</code> DOUBLE COMPLEX for <code>zhetf2</code> . Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the Hermitian matrix A . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$.

Output Parameters

<i>a</i>	On exit, the block diagonal matrix D and the multipliers used to obtain the factor U or L .
<i>ipiv</i>	INTEGER. Array, DIMENSION (<i>n</i>). Details of the interchanges and the block structure of D If <i>ipiv</i> (<i>k</i>) > 0, then rows and columns <i>k</i> and <i>ipiv</i> (<i>k</i>) were interchanged and $D(k,k)$ is a 1-by-1 diagonal block. If <i>uplo</i> = 'U' and <i>ipiv</i> (<i>k</i>) = <i>ipiv</i> (<i>k-1</i>) < 0, then rows and columns <i>k-1</i> and - <i>ipiv</i> (<i>k</i>) were interchanged and $D(k-1:k, k-1:k)$ is a 2-by-2 diagonal block. If <i>uplo</i> = 'L' and <i>ipiv</i> (<i>k</i>) = <i>ipiv</i> (<i>k+1</i>) < 0, then rows and columns <i>k+1</i> and - <i>ipiv</i> (<i>k</i>) were interchanged and $D(k:k+1, k:k+1)$ is a 2-by-2 diagonal block.
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = - <i>k</i> , the <i>k</i> -th argument had an illegal value

> 0: if $\text{info} = k$, $D(k,k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?hetf2_rook

Computes the factorization of a complex Hermitian matrix, using the bounded Bunch-Kaufman diagonal pivoting method (unblocked algorithm).

Syntax

```
call chetf2_rook( uplo, n, a, lda, ipiv, info )
call zhetf2_rook( uplo, n, a, lda, ipiv, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine computes the factorization of a complex Hermitian matrix A using the bounded Bunch-Kaufman ("rook") diagonal pivoting method:

$$A = U^* D^* U^H \text{ or } A = L^* D^* L^H$$

where U (or L) is a product of permutation and unit upper (lower) triangular matrices, U^H is the conjugate transpose of U , and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the unblocked version of the algorithm, calling [BLAS Level 2 Routines](#).

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>a</i>	COMPLEX for chetf2_rook DOUBLE COMPLEX for zhetf2_rook. Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the Hermitian matrix A . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a On exit, the block diagonal matrix *D* and the multipliers used to obtain the factor *U* or *L*.

ipiv INTEGER. Array, DIMENSION (*n*).
Details of the interchanges and the block structure of *D*.
If $ipiv(k) > 0$, then rows and columns *k* and $ipiv(k)$ were interchanged and $D(k,k)$ is a 1-by-1 diagonal block.
If $uplo = 'U'$ and $ipiv(k) < 0$ and $ipiv(k - 1) < 0$, then rows and columns *k* and $-ipiv(k)$ were interchanged, rows and columns *k* - 1 and $-ipiv(k - 1)$ were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.
If $uplo = 'L'$ and $ipiv(k) < 0$ and $ipiv(k + 1) < 0$, then rows and columns *k* and $-ipiv(k)$ were interchanged, rows and columns *k* + 1 and $-ipiv(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

info INTEGER.
= 0: successful exit
< 0: if *info* = $-k$, the *k*-th argument had an illegal value
> 0: if *info* = *k*, $D(k,k)$ is exactly zero. The factorization has been completed, but the block diagonal matrix *D* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?tgex2

Swaps adjacent diagonal blocks in an upper (quasi) triangular matrix pair by an orthogonal/unitary equivalence transformation.

Syntax

```
call stgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1, n2, work,
lwork, info )
call dtgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, n1, n2, work,
lwork, info )
call ctgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, info )
call ztgex2( wantq, wantz, n, a, lda, b, ldb, q, ldq, z, ldz, j1, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The real routines stgex2/dtgex2 swap adjacent diagonal blocks (A_{11}, B_{11}) and (A_{22}, B_{22}) of size 1-by-1 or 2-by-2 in an upper (quasi) triangular matrix pair (*A*, *B*) by an orthogonal equivalence transformation. (*A*, *B*) must be in generalized real Schur canonical form (as returned by sgges/dgges), that is, *A* is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks. *B* is upper triangular.

The complex routines `ctgex2/ztgex2` swap adjacent diagonal 1-by-1 blocks (A_{11}, B_{11}) and (A_{22}, B_{22}) in an upper triangular matrix pair (A, B) by an unitary equivalence transformation.

(A, B) must be in generalized Schur canonical form, that is, A and B are both upper triangular.

All routines optionally update the matrices Q and Z of generalized Schur vectors:

For real flavors,

$$Q(\text{in}) * A(\text{in}) * Z(\text{in})^T = Q(\text{out}) * A(\text{out}) * Z(\text{out})^T$$

$$Q(\text{in}) * B(\text{in}) * Z(\text{in})^T = Q(\text{out}) * B(\text{out}) * Z(\text{out})^T.$$

For complex flavors,

$$Q(\text{in}) * A(\text{in}) * Z(\text{in})^H = Q(\text{out}) * A(\text{out}) * Z(\text{out})^H$$

$$Q(\text{in}) * B(\text{in}) * Z(\text{in})^H = Q(\text{out}) * B(\text{out}) * Z(\text{out})^H.$$

Input Parameters

<code>wantq</code>	LOGICAL. If <code>wantq</code> = .TRUE. : update the left transformation matrix Q ; If <code>wantq</code> = .FALSE. : do not update Q .
<code>wantz</code>	LOGICAL. If <code>wantz</code> = .TRUE. : update the right transformation matrix Z ; If <code>wantz</code> = .FALSE. : do not update Z .
<code>n</code>	INTEGER. The order of the matrices A and B . $n \geq 0$.
<code>a, b</code>	REAL for <code>stgex2</code> DOUBLE PRECISION for <code>dtgex2</code> COMPLEX for <code>ctgex2</code> DOUBLE COMPLEX for <code>ztgex2</code> . Arrays, DIMENSION (<code>lda, n</code>) and (<code>ldb, n</code>), respectively. On entry, the matrices A and B in the pair (A, B).
<code>lda</code>	INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.
<code>ldb</code>	INTEGER. The leading dimension of the array b . $ldb \geq \max(1, n)$.
<code>q, z</code>	REAL for <code>stgex2</code> DOUBLE PRECISION for <code>dtgex2</code> COMPLEX for <code>ctgex2</code> DOUBLE COMPLEX for <code>ztgex2</code> . Arrays, DIMENSION (<code>ldq, n</code>) and (<code>ldz, n</code>), respectively. On entry, if <code>wantq</code> = .TRUE., q contains the orthogonal/unitary matrix Q , and if <code>wantz</code> = .TRUE., z contains the orthogonal/unitary matrix Z .
<code>ldq</code>	INTEGER. The leading dimension of the array q . $ldq \geq 1$. If <code>wantq</code> = .TRUE., $ldq \geq n$.
<code>ldz</code>	INTEGER. The leading dimension of the array z . $ldz \geq 1$. If <code>wantz</code> = .TRUE., $ldz \geq n$.

<i>j1</i>	INTEGER. The index to the first block (A_{11}, B_{11}). $1 \leq j1 \leq n$.
<i>n1</i>	INTEGER. Used with real flavors only. The order of the first block (A_{11}, B_{11}). $n1 = 0, 1$ or 2 .
<i>n2</i>	INTEGER. Used with real flavors only. The order of the second block (A_{22}, B_{22}). $n2 = 0, 1$ or 2 .
<i>work</i>	REAL for <code>stgex2</code> DOUBLE PRECISION for <code>dtgex2</code> . Workspace array, DIMENSION (<code>max(1, lwork)</code>). Used with real flavors only.
<i>lwork</i>	INTEGER. The dimension of the array <i>work</i> . $lwork \geq \max(n * (n2 + n1), 2 * (n2 + n1)^2)$

Output Parameters

<i>a</i>	On exit, the updated matrix A .
<i>B</i>	On exit, the updated matrix B .
<i>Q</i>	On exit, the updated matrix Q . Not referenced if <code>wantq = .FALSE..</code>
<i>Z</i>	On exit, the updated matrix Z . Not referenced if <code>wantz = .FALSE..</code>
<i>info</i>	INTEGER. =0: Successful exit For <code>stgex2/dtgex2</code> : If <code>info = 1</code> , the transformed matrix (A, B) would be too far from generalized Schur form; the blocks are not swapped and (A, B) and (Q, Z) are unchanged. The problem of swapping is too ill-conditioned. If <code>info = -16</code> : <i>lwork</i> is too small. Appropriate value for <i>lwork</i> is returned in <i>work(1)</i> . For <code>ctgex2/ztgex2</code> : If <code>info = 1</code> , the transformed matrix pair (A, B) would be too far from generalized Schur form; the problem is ill-conditioned.

?tgsy2

Solves the generalized Sylvester equation (unblocked algorithm).

Syntax

```
call stgex2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, rdsum, rdscal, iwork, pq, info )
call dtgex2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, rdsum, rdscal, iwork, pq, info )
```

```
call ctgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, rdsum, rdscal, iwork, pq, info )
call ztgsy2( trans, ijob, m, n, a, lda, b, ldb, c, ldc, d, ldd, e, lde, f, ldf,
scale, rdsum, rdscal, iwork, pq, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?tgssy2 solves the generalized Sylvester equation:

$$A^*R - L^*B = scale*C \quad (1)$$

$$D^*R - L^*E = scale*F$$

using Level 1 and 2 BLAS, where R and L are unknown m -by- n matrices, (A, D) , (B, E) and (C, F) are given matrix pairs of size m -by- m , n -by- n and m -by- n , respectively. For stgssy2/dtgssy2, pairs (A, D) and (B, E) must be in generalized Schur canonical form, that is, A, B are upper quasi triangular and D, E are upper triangular. For ctgssy2/ztgssy2, matrices A, B, D and E are upper triangular (that is, (A, D) and (B, E) in generalized Schur form).

The solution (R, L) overwrites (C, F) .

$0 \leq scale \leq 1$ is an output scaling factor chosen to avoid overflow.

In matrix notation, solving equation (1) corresponds to solve

$$Z^*x = scale*b$$

where Z is defined for real flavors as

$$Z = \begin{bmatrix} \text{kron}(I_n, A) & -\text{kron}(B^T, I_n) \\ \text{kron}(I_n, D) & -\text{kron}(E^T, I_n) \end{bmatrix} \quad (2)$$

and for complex flavors as

$$Z = \begin{bmatrix} \text{kron}(I_n, A) & -\text{kron}(B^H, I_n) \\ \text{kron}(I_n, D) & -\text{kron}(E^H, I_n) \end{bmatrix} \quad (3)$$

Here I_k is the identity matrix of size k and X^T (X^H) is the transpose (conjugate transpose) of X . $\text{kron}(X, Y)$ denotes the Kronecker product between the matrices X and Y .

For real flavors, if $trans = 'T'$, solve the transposed system

$$Z^T y = scale*b$$

for y , which is equivalent to solving for R and L in

$$A^T R + D^T L = scale*C \quad (4)$$

$$R^T B^T + L^T E^T = scale*(-F)$$

For complex flavors, if $trans = 'C'$, solve the conjugate transposed system

$$Z^H y = scale*b$$

for y , which is equivalent to solving for R and L in

$$A^H * R + D^H * L = \text{scale} * C \quad (5)$$

$$R^* B^H + L^* E^H = \text{scale} * (-F)$$

These cases are used to compute an estimate of $\text{Dif}[(A, D), (B, E)] = \sigma_{\min}(Z)$ using reverse communication with [?lacon](#).

?tgpsy2 also (for $i_{\text{job}} \geq 1$) contributes to the computation in ?tgtsy1 of an upper bound on the separation between two matrix pairs. Then the input (A, D) , (B, E) are sub-pencils of the matrix pair (two matrix pairs) in ?tgtsy1. See [?tgtsy1](#) for details.

Input Parameters

<i>trans</i>	CHARACTER*1. If <i>trans</i> = 'N', solve the generalized Sylvester equation (1); If <i>trans</i> = 'T': solve the transposed system (4). If <i>trans</i> = 'C': solve the conjugate transposed system (5).
<i>ijob</i>	INTEGER. Specifies what kind of functionality is to be performed. If <i>ijob</i> = 0: solve (1) only. If <i>ijob</i> = 1: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (look ahead strategy is used); If <i>ijob</i> = 2: a contribution from this subsystem to a Frobenius norm-based estimate of the separation between two matrix pairs is computed (?gecon on sub-systems is used). Not referenced if <i>trans</i> = 'T'.
<i>m</i>	INTEGER. On entry, <i>m</i> specifies the order of A and D , and the row dimension of C , F , R and L .
<i>n</i>	INTEGER. On entry, <i>n</i> specifies the order of B and E , and the column dimension of C , F , R and L .
<i>a</i> , <i>b</i>	REAL for stgpsy2 DOUBLE PRECISION for dtgpsy2 COMPLEX for ctgpsy2 DOUBLE COMPLEX for ztgpsy2. Arrays, DIMENSION (<i>lda</i> , <i>m</i>) and (<i>ldb</i> , <i>n</i>), respectively. On entry, <i>a</i> contains an upper (quasi) triangular matrix A , and <i>b</i> contains an upper (quasi) triangular matrix B .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . <i>lda</i> $\geq \max(1, m)$.
<i>ldb</i>	INTEGER. The leading dimension of the array <i>b</i> . <i>ldb</i> $\geq \max(1, n)$.
<i>c</i> , <i>f</i>	REAL for stgpsy2 DOUBLE PRECISION for dtgpsy2

COMPLEX for `ctgsy2`

DOUBLE COMPLEX for `ztgsy2`.

Arrays, DIMENSION (ldc, n) and (ldf, n), respectively. On entry, c contains the right-hand-side of the first matrix equation in (1), and f contains the right-hand-side of the second matrix equation in (1).

ldc INTEGER. The leading dimension of the array c . $ldc \geq \max(1, m)$.

d, e REAL for `stgsy2`

DOUBLE PRECISION for `dtgsy2`

COMPLEX for `ctgsy2`

DOUBLE COMPLEX for `ztgsy2`.

Arrays, DIMENSION (lde, n) and (ldf, n), respectively. On entry, d contains an upper triangular matrix D , and e contains an upper triangular matrix E .

lde INTEGER. The leading dimension of the array d . $lde \geq \max(1, n)$.

ldf INTEGER. The leading dimension of the array f . $ldf \geq \max(1, m)$.

$rdsum$ REAL for `stgsy2/ctgsy2`

DOUBLE PRECISION for `dtgsy2/ztgsy2`.

On entry, the sum of squares of computed contributions to the `Dif-` estimate under computation by `?tgsyL`, where the scaling factor $rdscal$ has been factored out.

$rdscal$ REAL for `stgsy2/ctgsy2`

DOUBLE PRECISION for `dtgsy2/ztgsy2`.

On entry, scaling factor used to prevent overflow in $rdsum$.

$iwork$ INTEGER. Used with real flavors only.

Workspace array, DIMENSION ($m+n+2$).

Output Parameters

c On exit, if $ijob = 0$, c is overwritten by the solution R .

f On exit, if $ijob = 0$, f is overwritten by the solution L .

$scale$ REAL for `stgsy2/ctgsy2`

DOUBLE PRECISION for `dtgsy2/ztgsy2`.

On exit, $0 \leq scale \leq 1$. If $0 < scale < 1$, the solutions R and L (C and F on entry) hold the solutions to a slightly perturbed system, but the input matrices A, B, D and E are not changed. If $scale = 0$, R and L hold the solutions to the homogeneous system with $C = F = 0$. Normally $scale = 1$.

<i>rdsum</i>	On exit, the corresponding sum of squares updated with the contributions from the current sub-system. If <i>trans</i> = 'T', <i>rdsum</i> is not touched.
	Note that <i>rdsum</i> only makes sense when ?tgsy2 is called by ?tgsvl.
<i>rdscal</i>	On exit, <i>rdscal</i> is updated with respect to the current contributions in <i>rdsum</i> . If <i>trans</i> = 'T', <i>rdscal</i> is not touched.
	Note that <i>rdscal</i> only makes sense when ?tgsy2 is called by ?tgsvl.
<i>pq</i>	INTEGER. Used with real flavors only. On exit, the number of subsystems (of size 2-by-2, 4-by-4 and 8-by-8) solved by the routine stgsy2/dtgsy2.
<i>info</i>	INTEGER. On exit, if <i>info</i> is set to = 0: Successful exit < 0: If <i>info</i> = - <i>i</i> , the <i>i</i> -th argument has an illegal value. > 0: The matrix pairs (<i>A</i> , <i>D</i>) and (<i>B</i> , <i>E</i>) have common or very close eigenvalues.

?trti2

Computes the inverse of a triangular matrix
(unblocked algorithm).

Syntax

```
call strtti2( uplo, diag, n, a, lda, info )
call dtrtti2( uplo, diag, n, a, lda, info )
call ctrtti2( uplo, diag, n, a, lda, info )
call ztrtti2( uplo, diag, n, a, lda, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?trti2 computes the inverse of a real/complex upper or lower triangular matrix.

This is the *Level 2 BLAS* version of the algorithm.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular. = 'U': upper triangular = 'L': lower triangular
-------------	---

diag CHARACTER*1.
 Specifies whether or not the matrix A is unit triangular.
 = 'N': non-unit triangular
 = 'U': non-unit triangular

n INTEGER. The order of the matrix A . $n \geq 0$.

a REAL for strti2
 DOUBLE PRECISION for dtrti2
 COMPLEX for ctrti2
 DOUBLE COMPLEX for ztrti2.
 Array, DIMENSION (*lda*, *n*).
 On entry, the triangular matrix A .
 If *uplo* = 'U', the leading n -by- n upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced.
 If *uplo* = 'L', the leading n -by- n lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced.
 If *diag* = 'U', the diagonal elements of *a* are also not referenced and are assumed to be 1.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

Output Parameters

a On exit, the (triangular) inverse of the original matrix, in the same storage format.

info INTEGER.
 = 0: successful exit
 < 0: if *info* = $-k$, the k -th argument had an illegal value

clag2z

Converts a complex single precision matrix to a complex double precision matrix.

Syntax

call clag2z(*m*, *n*, *sa*, *ldsa*, *a*, *lda*, *info*)

C:

```
lapack_int LAPACKE_clag2z (int matrix_layout, lapack_int m, lapack_int n, const
lapack_complex_float * sa, lapack_int ldsa, lapack_complex_double * a, lapack_int lda);
```

Include Files

- Fortran: mkl.fi

-
- C: mkl.h

Description

This routine converts a complex single precision matrix SA to a complex double precision matrix A .

Note that while it is possible to overflow while converting from double to single, it is not possible to overflow when converting from single to double.

This is an auxiliary routine so there is no argument checking.

Input Parameters

<i>m</i>	INTEGER. The number of lines of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix A ($n \geq 0$).
<i>ldsa</i>	INTEGER. The leading dimension of the array sa ; $ldsa \geq \max(1, m)$.
<i>a</i>	DOUBLE PRECISION array, DIMENSION (lda, n). On entry, contains the m -by- n coefficient matrix A .
<i>lda</i>	INTEGER. The leading dimension of the array a ; $lda \geq \max(1, m)$.

Output Parameters

<i>sa</i>	REAL array, DIMENSION ($ldsa, n$). On exit, contains the m -by- n coefficient matrix SA .
<i>info</i>	INTEGER. If $info = 0$, the execution is successful.

dlag2s

Converts a double precision matrix to a single precision matrix.

Syntax

```
call dlag2s( m, n, a, lda, sa, ldsa, info )
```

C:

```
lapack_int LAPACKE_dlag2s (int matrix_layout, lapack_int m, lapack_int n, const double * a, lapack_int lda, float * sa, lapack_int ldsa);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This routine converts a double precision matrix SA to a single precision matrix A .

RMAX is the overflow for the single precision arithmetic. dlag2s checks that all the entries of A are between -RMAX and RMAX. If not, the conversion is aborted and a flag is raised.

This is an auxiliary routine so there is no argument checking.

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix A ($n \geq 0$).
<i>a</i>	DOUBLE PRECISION array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, contains the m -by- n coefficient matrix A .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> ; <i>lda</i> $\geq \max(1, m)$.
<i>ldsa</i>	INTEGER. The leading dimension of the array <i>sa</i> ; <i>ldsa</i> $\geq \max(1, m)$.

Output Parameters

<i>sa</i>	REAL array, DIMENSION (<i>ldsa</i> , <i>n</i>). On exit, if <i>info</i> = 0, contains the m -by- n coefficient matrix SA ; if <i>info</i> > 0, the content of <i>sa</i> is unspecified.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = 1, an entry of the matrix A is greater than the single precision overflow threshold; in this case, the content of <i>sa</i> on exit is unspecified.

slag2d

Converts a single precision matrix to a double precision matrix.

Syntax

```
call slag2d( m, n, sa, ldsa, a, lda, info )
lapack_int LAPACKE_slag2d (int matrix_layout, lapack_int m, lapack_int n, const float * sa, lapack_int ldsa, double * a, lapack_int lda);
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine converts a single precision matrix SA to a double precision matrix A .

Note that while it is possible to overflow while converting from double to single, it is not possible to overflow when converting from single to double.

This is an auxiliary routine so there is no argument checking.

Input Parameters

<i>m</i>	INTEGER. The number of lines of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in the matrix A ($n \geq 0$).

sa REAL array, DIMENSION (*ldsa*, *n*).
 On entry, contains the *m*-by-*n* coefficient matrix *SA*.

ldsa INTEGER. The leading dimension of the array *sa*; *ldsa* $\geq \max(1, m).$

lda INTEGER. The leading dimension of the array *a*; *lda* $\geq \max(1, m).$

Output Parameters

a DOUBLE PRECISION array, DIMENSION (*lda*, *n*).
 On exit, contains the *m*-by-*n* coefficient matrix *A*.

info INTEGER.
 If *info* = 0, the execution is successful.

zlag2c

Converts a complex double precision matrix to a complex single precision matrix.

Syntax

```
call zlag2c( m, n, a, lda, sa, ldsa, info )
```

C:

```
lapack_int LAPACKE_zlag2c (int matrix_layout, lapack_int m, lapack_int n, const  

                           lapack_complex_double * a, lapack_int lda, lapack_complex_float * sa, lapack_int ldsa);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine converts a double precision complex matrix *SA* to a single precision complex matrix *A*.

RMAX is the overflow for the single precision arithmetic. zlag2c checks that all the entries of *A* are between -RMAX and RMAX. If not, the conversion is aborted and a flag is raised.

This is an auxiliary routine so there is no argument checking.

Input Parameters

m INTEGER. The number of lines of the matrix *A* (*m* $\geq 0).
n INTEGER. The number of columns in the matrix *A* (*n* ≥ 0).
a DOUBLE COMPLEX array, DIMENSION (*lda*, *n*).
 On entry, contains the *m*-by-*n* coefficient matrix *A*.
lda INTEGER. The leading dimension of the array *a*; *lda* $\geq \max(1, m).
ldsa INTEGER. The leading dimension of the array *sa*; *ldsa* $\geq \max(1, m).$$$

Output Parameters

*sa*COMPLEX array, DIMENSION (*ldsa*, *n*).

On exit, if *info* = 0, contains the *m*-by-*n* coefficient matrix *SA*; if *info* > 0, the content of *sa* is unspecified.

info

INTEGER.

If *info* = 0, the execution is successful.

If *info* = 1, an entry of the matrix *A* is greater than the single precision overflow threshold; in this case, the content of *sa* on exit is unspecified.

?larfp

Generates a real or complex elementary reflector.

Syntax

FORTRAN 77:

```
call slarfp(n, alpha, x, incx, tau)
call dlarfp(n, alpha, x, incx, tau)
call clarfp(n, alpha, x, incx, tau)
call zlarfp(n, alpha, x, incx, tau)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?larfp routines generate a real or complex elementary reflector *H* of order *n*, such that

```
H * (alpha) = (beta),
( x )   ( 0 )
```

and $H^*H = I$ for real flavors, $\text{conjg}(H)^*H = I$ for complex flavors.

Here

alpha and *beta* are scalars, *beta* is real and non-negative,

x is (*n*-1)-element vector.

H is represented in the form

```
H = I - tau* ( 1 ) * ( 1 v ),
( v )
```

where *tau* is scalar, and *v* is (*n*-1)-element vector .

For real flavors if the elements of *x* are all zero, then *tau* = 0 and *H* is taken to be the unit matrix. Otherwise $1 \leq \text{real}(\tau) \leq 2$.

For complex flavors if the elements of *x* are all zero and *alpha* is real, then *tau* = 0 and *H* is taken to be the unit matrix. Otherwise $1 \leq \text{real}(\tau) \leq 2$, and $\text{abs}(\tau) \leq 1$.

Input Parameters

<i>n</i>	INTEGER. Specifies the order of the elementary reflector.
<i>alpha</i>	REAL for slarf _p DOUBLE PRECISION for dlarf _p COMPLEX for clarf _p DOUBLE COMPLEX for zlarf _p Specifies the scalar <i>alpha</i> .
<i>x</i>	REAL for slarf _p DOUBLE PRECISION for dlarf _p COMPLEX for clarf _p DOUBLE COMPLEX for zlarf _p Array, DIMENSION at least (1 + (n - 1)*abs(<i>incx</i>)). It contains the vector <i>x</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must not be zero.

Output Parameters

<i>alpha</i>	Overwritten by the value <i>beta</i> .
<i>y</i>	Overwritten by the vector <i>v</i> .
<i>tau</i>	REAL for slarf _p DOUBLE PRECISION for dlarf _p COMPLEX for clarf _p DOUBLE COMPLEX for zlarf _p Contains the scalar <i>tau</i> .

ila?lc

Scans a matrix for its last non-zero column.

Syntax

FORTRAN 77:

```
value = ilaslc(m, n, a, lda)
value = iladlc(m, n, a, lda)
value = ilaclc(m, n, a, lda)
value = ilazlc(m, n, a, lda)
```

Include Files

- Fortran: mkl.fi

-
- C: mkl.h

Description

The ila?lc routines scan a matrix A for its last non-zero column.

Input Parameters

<i>m</i>	INTEGER. Specifies number of rows in the matrix A .
<i>n</i>	INTEGER. Specifies number of columns in the matrix A .
<i>a</i>	REAL for ilaslc DOUBLE PRECISION for iladlc COMPLEX for ilaclc DOUBLE COMPLEX for ilazlc Array, DIMENSION (<i>lda</i> , *). The second dimension of <i>a</i> must be at least $\max(1, n)$. Before entry the leading n -by- n part of the array <i>a</i> must contain the matrix A .
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, m)$.

Output Parameters

<i>value</i>	INTEGER Number of the last non-zero column.
--------------	--

ila?lr

Scans a matrix for its last non-zero row.

Syntax

FORTRAN 77:

```
value = ilaslr(m, n, a, lda)
value = iladlr(m, n, a, lda)
value = ilaclr(m, n, a, lda)
value = ilazlr(m, n, a, lda)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ila?lr routines scan a matrix A for its last non-zero row.

Input Parameters

<i>m</i>	INTEGER. Specifies number of rows in the matrix <i>A</i> .
<i>n</i>	INTEGER. Specifies number of columns in the matrix <i>A</i> .
<i>a</i>	REAL for <code>ilaslr</code> DOUBLE PRECISION for <code>iladlr</code> COMPLEX for <code>ilaclr</code> DOUBLE COMPLEX for <code>idazlr</code> Array, DIMENSION (<i>lda</i> , *). The second dimension of <i>a</i> must be at least <code>max(1, n)</code> . Before entry the leading <i>n</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix <i>A</i> .
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least <code>max(1, m)</code> .

Output Parameters

<i>value</i>	INTEGER Number of the last non-zero row.
--------------	---

?gsvj0

Pre-processor for the routine ?gesvj.

Syntax

FORTRAN 77:

```
call sgsvj0(jobv, m, n, a, lda, d, sva, mv, v, ldv, eps, sfmin, tol, nsweep, work, lwork, info)
call dgsvj0(jobv, m, n, a, lda, d, sva, mv, v, ldv, eps, sfmin, tol, nsweep, work, lwork, info)
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

This routine is called from `?gesvj` as a pre-processor and that is its main purpose. It applies Jacobi rotations in the same way as `?gesvj` does, but it does not check convergence (stopping criterion).

The routine `?gsvj0` enables `?gesvj` to use a simplified version of itself to work on a submatrix of the original matrix.

Input Parameters

<i>jobv</i>	CHARACTER*1. Must be 'V', 'A', or 'N'.
-------------	--

Specifies whether the output from this routine is used to compute the matrix V .

If $jobv = 'V'$, the product of the Jacobi rotations is accumulated by post-multiplying the n -by- n array v .

If $jobv = 'A'$, the product of the Jacobi rotations is accumulated by post-multiplying the mv -by- n array v .

If $jobv = 'N'$, the Jacobi rotations are not accumulated.

m INTEGER. The number of rows of the input matrix A ($m \geq 0$).

n INTEGER. The number of columns of the input matrix B ($m \geq n \geq 0$).

a REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

Arrays, DIMENSION ($lda, *$). Contains the m -by- n matrix A , such that $A^*\text{diag}(D)$ represents the input matrix. The second dimension of a must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of a ; at least $\max(1, m)$.

d REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

Arrays, DIMENSION (n). Contains the diagonal matrix D that accumulates the scaling factors from the fast scaled Jacobi rotations. On entry $A^*\text{diag}(D)$ represents the input matrix.

sva REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

Arrays, DIMENSION (n). Contains the Euclidean norms of the columns of the matrix $A^*\text{diag}(D)$.

mv INTEGER. The leading dimension of b ; at least $\max(1, p)$.

If $jobv = 'A'$, then mv rows of v are post-multiplied by a sequence of Jacobi rotations.

If $jobv = 'N'$, then mv is not referenced .

v REAL for sgsvj0

DOUBLE PRECISION for dgsvj0.

Array, DIMENSION ($ldv, *$). The second dimension of a must be at least $\max(1, n)$.

If $jobv = 'V'$, then n rows of v are post-multiplied by a sequence of Jacobi rotations.

If $jobv = 'A'$, then mv rows of v are post-multiplied by a sequence of Jacobi rotations.

If $jobv = 'N'$, then v is not referenced.

ldv INTEGER. The leading dimension of the array v ; $ldv \geq 1$

```

 $ldv \geq n$  if  $jobv = 'V'$ ;  

 $ldv \geq mv$  if  $jobv = 'A'$ .  

 $eps$  REAL for sgsvj0  

    DOUBLE PRECISION for dgsvj0.  

    The relative machine precision (epsilon) returned by the routine ?lamch.  

 $sfmin$  REAL for sgsvj0  

    DOUBLE PRECISION for dgsvj0.  

    Value of safe minimum returned by the routine ?lamch.  

 $tol$  REAL for sgsvj0  

    DOUBLE PRECISION for dgsvj0.  

    The threshold for Jacobi rotations. For a pair  $A(:, p), A(:, q)$  of pivot  

    columns, the Jacobi rotation is applied only if  $\text{abs}(\cos(\text{angle}(A(:, p),  

    A(:, q)))) > tol$ .  

 $nsweep$  INTEGER.  

    The number of sweeps of Jacobi rotations to be performed.  

 $work$  REAL for sgsvj0  

    DOUBLE PRECISION for dgsvj0.  

    Workspace array, DIMENSION ( $lwork$ ).  

 $lwork$  INTEGER. The size of the array  $work$ ; at least  $\max(1, m)$ .

```

Output Parameters

a	On exit, $A^*\text{diag}(D)$ represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in tol and $nsweep$, respectively
d	On exit, $A^*\text{diag}(D)$ represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in tol and $nsweep$, respectively.
sva	On exit, contains the Euclidean norms of the columns of the output matrix $A^*\text{diag}(D)$.
v	If $jobv = 'V'$, then n rows of v are post-multiplied by a sequence of Jacobi rotations. If $jobv = 'A'$, then mv rows of v are post-multiplied by a sequence of Jacobi rotations. If $jobv = 'N'$, then v is not referenced.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info = -i$, the i -th parameter had an illegal value.

?gsvj1

Pre-processor for the routine ?gesvj, applies Jacobi rotations targeting only particular pivots.

Syntax

FORTRAN 77:

```
call sgsvj1(jobv, m, n, n1, a, lda, d, sva, mv, v, ldv, eps, sfmin, tol, nsweep,
work, lwork, info)
```

```
call dgsvj1(jobv, m, n, n1, a, lda, d, sva, mv, v, ldv, eps, sfmin, tol, nsweep,
work, lwork, info)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

T

This routine is called from ?gesvj as a pre-processor and that is its main purpose. It applies Jacobi rotations in the same way as ?gesvj does, but it targets only particular pivots and it does not check convergence (stopping criterion).

The routine ?gsvj1 applies few sweeps of Jacobi rotations in the column space of the input m -by- n matrix A . The pivot pairs are taken from the (1,2) off-diagonal block in the corresponding n -by- n Gram matrix $A^* A$. The block-entries (*tiles*) of the (1,2) off-diagonal block are marked by the [x]'s in the following scheme:

	*	*	*	[x]	[x]	[x]	
	*	*	*	[x]	[x]	[x]	
	*	*	*	[x]	[x]	[x]	
	[x]	[x]	[x]	*	*	*	
	[x]	[x]	[x]	*	*	*	
	[x]	[x]	[x]	*	*	*	

row-cycling in the $nblr$ -by- $nblc$ [x] blocks, row-cyclic pivoting inside each [x] block

In terms of the columns of the matrix A , the first $n1$ columns are rotated 'against' the remaining $n-n1$ columns, trying to increase the angle between the corresponding subspaces. The off-diagonal block is $n1$ -by- $(n-n1)$ and it is tiled using quadratic tiles. The number of sweeps is specified by $nsweep$, and the orthogonality threshold is set by tol .

Input Parameters

jobv

CHARACTER*1. Must be 'V', 'A', or 'N'.

Specifies whether the output from this routine is used to compute the matrix V .

If $jobv = 'V'$, the product of the Jacobi rotations is accumulated by post-multiplying the n -by- n array v .

If $jobv = 'A'$, the product of the Jacobi rotations is accumulated by post-multiplying the mv -by- n array v .

If $jobv = 'N'$, the Jacobi rotations are not accumulated.

m

INTEGER. The number of rows of the input matrix A ($m \geq 0$).

n INTEGER. The number of columns of the input matrix *B* ($m \geq n \geq 0$).

n1 INTEGER. Specifies the 2-by-2 block partition. The first *n1* columns are rotated 'against' the remaining *n-n1* columns of the matrix *A*.

a REAL for sgsvj1
DOUBLE PRECISION for dgsvj1.
Arrays, DIMENSION (*lda*, *). Contains the m -by- n matrix *A*, such that $A^*\text{diag}(D)$ represents the input matrix. The second dimension of *a* must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of *a*; at least $\max(1, m)$.

d REAL for sgsvj1
DOUBLE PRECISION for dgsvj1.
Arrays, DIMENSION (*n*). Contains the diagonal matrix *D* that accumulates the scaling factors from the fast scaled Jacobi rotations. On entry $A^*\text{diag}(D)$ represents the input matrix.

sva REAL for sgsvj1
DOUBLE PRECISION for dgsvj1.
Arrays, DIMENSION (*n*). Contains the Euclidean norms of the columns of the matrix $A^*\text{diag}(D)$.

mv INTEGER. The leading dimension of *b*; at least $\max(1, p)$.
If *jobv* = 'A', then *mv* rows of *v* are post-multiplied by a sequence of Jacobi rotations.
If *jobv* = 'N', then *mv* is not referenced .

v REAL for sgsvj1
DOUBLE PRECISION for dgsvj1.
Array, DIMENSION (*ldv*, *). The second dimension of *a* must be at least $\max(1, n)$.
If *jobv* = 'V', then *n* rows of *v* are post-multiplied by a sequence of Jacobi rotations.
If *jobv* = 'A', then *mv* rows of *v* are post-multiplied by a sequence of Jacobi rotations.
If *jobv* = 'N', then *v* is not referenced.

ldv INTEGER. The leading dimension of the array *v*; *ldv* ≥ 1
ldv $\geq n$ if *jobv* = 'V';
ldv $\geq mv$ if *jobv* = 'A'.

eps REAL for sgsvj1
DOUBLE PRECISION for dgsvj1.
The relative machine precision (epsilon) returned by the routine [?lamch](#).

<i>sfmin</i>	REAL for sgsvj1 DOUBLE PRECISION for dgsvj1. Value of safe minimum returned by the routine ?lamch.
<i>tol</i>	REAL for sgsvj1 DOUBLE PRECISION for dgsvj1. The threshold for Jacobi rotations. For a pair $A(:, p), A(:, q)$ of pivot columns, the Jacobi rotation is applied only if $\text{abs}(\cos(\text{angle}(A(:, p), A(:, q)))) > tol$.
<i>nsweep</i>	INTEGER. The number of sweeps of Jacobi rotations to be performed.
<i>work</i>	REAL for sgsvj1 DOUBLE PRECISION for dgsvj1. Workspace array, DIMENSION (<i>lwork</i>).
<i>lwork</i>	INTEGER. The size of the array <i>work</i> ; at least $\max(1, m)$.

Output Parameters

<i>a</i>	On exit, $A^*\text{diag}(D)$ represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in <i>tol</i> and <i>nsweep</i> , respectively
<i>d</i>	On exit, $A^*\text{diag}(D)$ represents the input matrix post-multiplied by a sequence of Jacobi rotations, where the rotation threshold and the total number of sweeps are given in <i>tol</i> and <i>nsweep</i> , respectively.
<i>sva</i>	On exit, contains the Euclidean norms of the columns of the output matrix $A^*\text{diag}(D)$.
<i>v</i>	If <i>jobv</i> = 'V', then <i>n</i> rows of <i>v</i> are post-multiplied by a sequence of Jacobi rotations. If <i>jobv</i> = 'A', then <i>mv</i> rows of <i>v</i> are post-multiplied by a sequence of Jacobi rotations. If <i>jobv</i> = 'N', then <i>v</i> is not referenced.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

?sfrk

Performs a symmetric rank-*k* operation for matrix in RFP format.

Syntax

FORTRAN 77:

```
call ssfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
call dsfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
```

C:

```
lapack_int LAPACKE_<?>sfrk( int matrix_layout, char transr, char uplo, char trans,
lapack_int n, lapack_int k, <datatype> alpha, const <datatype>* a, lapack_int lda,
<datatype> beta, <datatype>* c );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?sfrk routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

$$C := \alpha A^* A + \beta C,$$

or

$$C := \alpha A^T A + \beta C,$$

where:

alpha and *beta* are scalars,

C is an *n*-by-*n* symmetric matrix in [rectangular full packed \(RFP\) format](#),

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>transr</i>	CHARACTER*1. if <i>transr</i> = 'N' or 'n', the normal form of RFP <i>C</i> is stored; if <i>transr</i> = 'T' or 't', the transpose form of RFP <i>C</i> is stored.
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>c</i> is used.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then <i>C</i> := <i>alpha</i> * <i>A</i> * <i>A</i> T + <i>beta</i> * <i>C</i> ; if <i>trans</i> = 'T' or 't', then <i>C</i> := <i>alpha</i> * <i>A</i> T * <i>A</i> + <i>beta</i> * <i>C</i> ;
<i>n</i>	INTEGER. Specifies the order of the matrix <i>C</i> . The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the matrix <i>A</i> , and on entry with <i>trans</i> = 'T' or 't', <i>k</i> specifies the number of rows of the matrix <i>A</i> .

The value of k must be at least zero.

<i>alpha</i>	REAL for ssfrk DOUBLE PRECISION for dsfrk Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for ssfrk DOUBLE PRECISION for dsfrk Array, DIMENSION (<i>lda</i> , <i>ka</i>), where <i>ka</i> is k when <i>trans</i> = 'N' or 'n', and is n otherwise. Before entry with <i>trans</i> = 'N' or 'n', the leading n -by- k part of the array <i>a</i> must contain the matrix A , otherwise the leading k -by- n part of the array <i>a</i> must contain the matrix A .
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. When <i>trans</i> = 'N' or 'n', then <i>lda</i> must be at least $\max(1, n)$, otherwise <i>lda</i> must be at least $\max(1, k)$.
<i>beta</i>	REAL for ssfrk DOUBLE PRECISION for dsfrk Specifies the scalar <i>beta</i> .
<i>c</i>	REAL for ssfrk DOUBLE PRECISION for dsfrk Array, DIMENSION ($n * (n+1) / 2$). Before entry contains the symmetric matrix <i>C</i> in RFP format.

Output Parameters

<i>c</i>	If <i>trans</i> = 'N' or 'n', then <i>c</i> contains $C := \alpha A^* A + \beta C$; if <i>trans</i> = 'T' or 't', then <i>c</i> contains $C := \alpha A^* A^* + \beta C$;
----------	--

?hfrk

Performs a Hermitian rank- k operation for matrix in RFP format.

Syntax

FORTRAN 77:

```
call chfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
call zhfrk(transr, uplo, trans, n, k, alpha, a, lda, beta, c)
```

C:

```
lapack_int LAPACKE_chfrk( int matrix_layout, char transr, char uplo, char trans,
                           lapack_int n, lapack_int k, float alpha, const lapack_complex_float* a, lapack_int
                           lda, float beta, lapack_complex_float* c );
lapack_int LAPACKE_zhfrk( int matrix_layout, char transr, char uplo, char trans,
                           lapack_int n, lapack_int k, double alpha, const lapack_complex_double* a, lapack_int
                           lda, double beta, lapack_complex_double* c );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?hfrk routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

$$C := \alpha A^H A + \beta C,$$

or

$$C := \alpha A^H A + \beta C,$$

where:

alpha and *beta* are real scalars,

C is an *n*-by-*n* Hermitian matrix in [RFP format](#),

A is an *n*-by-*k* matrix in the first case and a *k*-by-*n* matrix in the second case.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>transr</i>	CHARACTER*1. if <i>transr</i> = 'N' or 'n', the normal form of RFP <i>C</i> is stored; if <i>transr</i> = 'C' or 'c', the conjugate-transpose form of RFP <i>C</i> is stored.
<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the array <i>c</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the array <i>c</i> is used.
<i>trans</i>	CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then <i>C</i> := $\alpha A^H A + \beta C$; if <i>trans</i> = 'C' or 'c', then <i>C</i> := $\alpha A^H A + \beta C$.
<i>n</i>	INTEGER. Specifies the order of the matrix <i>C</i> . The value of <i>n</i> must be at least zero.
<i>k</i>	INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the matrix <i>a</i> , and on entry with <i>trans</i> = 'T' or 't' or 'C' or 'c', <i>k</i> specifies the number of rows of the matrix <i>a</i> . The value of <i>k</i> must be at least zero.
<i>alpha</i>	COMPLEX for chfrk DOUBLE COMPLEX for zhfrk Specifies the scalar <i>alpha</i> .
<i>a</i>	COMPLEX for chfrk

DOUBLE COMPLEX for zhfrk

Array, DIMENSION (*lda, ka*), where *ka* is *k* when *trans* = 'N' or 'n', and is *n* otherwise. Before entry with *trans* = 'N' or 'n', the leading *n*-by-*k* part of the array *a* must contain the matrix *A*, otherwise the leading *k*-by-*n* part of the array *a* must contain the matrix *A*.

lda INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. When *trans* = 'N' or 'n', then *lda* must be at least $\max(1, n)$, otherwise *lda* must be at least $\max(1, k)$.

beta COMPLEX for chfrk
DOUBLE COMPLEX for zhfrk
Specifies the scalar *beta*.

c COMPLEX for chfrk
DOUBLE COMPLEX for zhfrk
Array, DIMENSION (*n*(n+1)/2*). Before entry contains the Hermitian matrix *C* in in [RFP format](#).

Output Parameters

c If *trans* = 'N' or 'n', then *c* contains $C := \alpha A^H + \beta C$;
if *trans* = 'C' or 'c', then *c* contains $C := \alpha A^H A + \beta C$;

?tfsm

Solves a matrix equation (one operand is a triangular matrix in RFP format).

Syntax

FORTRAN 77:

```
call stfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
call dtfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
call ctfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
call ztfsm(transr, side, uplo, trans, diag, m, n, alpha, a, b, ldb)
```

C:

```
lapack_int LAPACKE_<?>tfsm( int matrix_layout, char transr, char side, char uplo, char trans, char diag, lapack_int m, lapack_int n, <datatype> alpha, const <datatype>* a, <datatype>* b, lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?tfsm routines solve one of the following matrix equations:

*op(A)*X = alpha*B,*

or

$X^* \text{op}(A) = \alpha * B$,

where:

α is a scalar,

X and B are m -by- n matrices,

A is a unit, or non-unit, upper or lower triangular matrix in [rectangular full packed \(RFP\) format](#).

$\text{op}(A)$ can be one of the following:

- $\text{op}(A) = A$ or $\text{op}(A) = A^T$ for real flavors
- $\text{op}(A) = A$ or $\text{op}(A) = A^H$ for complex flavors

The matrix B is overwritten by the solution matrix X .

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>transr</i>	CHARACTER*1. if <i>transr</i> = 'N' or 'n', the normal form of RFP A is stored; if <i>transr</i> = 'T' or 't', the transpose form of RFP A is stored; if <i>transr</i> = 'C' or 'c', the conjugate-transpose form of RFP A is stored.
<i>side</i>	CHARACTER*1. Specifies whether $\text{op}(A)$ appears on the left or right of X in the equation: if <i>side</i> = 'L' or 'l', then $\text{op}(A)^*X = \alpha * B$; if <i>side</i> = 'R' or 'r', then $X^*\text{op}(A) = \alpha * B$.
<i>uplo</i>	CHARACTER*1. Specifies whether the RFP matrix A is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	CHARACTER*1. Specifies the form of $\text{op}(A)$ used in the matrix multiplication: if <i>trans</i> = 'N' or 'n', then $\text{op}(A) = A$; if <i>trans</i> = 'T' or 't', then $\text{op}(A) = A'$; if <i>trans</i> = 'C' or 'c', then $\text{op}(A) = \text{conjg}(A')$.
<i>diag</i>	CHARACTER*1. Specifies whether the RFP matrix A is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>m</i>	INTEGER. Specifies the number of rows of B . The value of <i>m</i> must be at least zero.

<i>n</i>	INTEGER. Specifies the number of columns of <i>B</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	<p>REAL for stfsm</p> <p>DOUBLE PRECISION for dtfsm</p> <p>COMPLEX for ctfsm</p> <p>DOUBLE COMPLEX for ztfsm</p> <p>Specifies the scalar <i>alpha</i>.</p> <p>When <i>alpha</i> is zero, then <i>a</i> is not referenced and <i>b</i> need not be set before entry.</p>
<i>a</i>	<p>REAL for stfsm</p> <p>DOUBLE PRECISION for dtfsm</p> <p>COMPLEX for ctfsm</p> <p>DOUBLE COMPLEX for ztfsm</p> <p>Array, DIMENSION ($n * (n+1) / 2$). Contains the matrix <i>A</i> in RFP format.</p>
<i>b</i>	<p>REAL for stfsm</p> <p>DOUBLE PRECISION for dtfsm</p> <p>COMPLEX for ctfsm</p> <p>DOUBLE COMPLEX for ztfsm</p> <p>Array, DIMENSION (ldb, n). Before entry, the leading <i>m</i>-by-<i>n</i> part of the array <i>b</i> must contain the right-hand side matrix <i>B</i>.</p>
<i>ldb</i>	INTEGER. Specifies the leading dimension of <i>b</i> as declared in the calling (sub)program. The value of <i>ldb</i> must be at least $\max(1, +m)$.

Output Parameters

<i>b</i>	Overwritten by the solution matrix <i>X</i> .
----------	---

?lansf

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a symmetric matrix in RFP format.

Syntax

```
val = slansf(norm, transr, uplo, n, a, work)
val = dlansf(norm, transr, uplo, n, a, work)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

T

The function `?lansf` returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n real symmetric matrix A in the [rectangular full packed \(RFP\) format](#).

Input Parameters

<code>norm</code>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': $val = \max(A_{ij})$, largest absolute value of the matrix A . = '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<code>transr</code>	CHARACTER*1. Specifies whether the RFP format of matrix A is normal or transposed format. If <code>transr</code> = 'N': RFP format is normal; if <code>transr</code> = 'T': RFP format is transposed.
<code>uplo</code>	CHARACTER*1. Specifies whether the RFP matrix A came from upper or lower triangular matrix. If <code>uplo</code> = 'U': RFP matrix A came from an upper triangular matrix; if <code>uplo</code> = 'L': RFP matrix A came from a lower triangular matrix.
<code>n</code>	INTEGER. The order of the matrix A . $n \geq 0$. When $n = 0$, <code>?lansf</code> is set to zero.
<code>a</code>	REAL for <code>slansf</code> DOUBLE PRECISION for <code>dlansf</code> Array, DIMENSION ($n*(n+1)/2$). The upper (if <code>uplo</code> = 'U') or lower (if <code>uplo</code> = 'L') part of the symmetric matrix A stored in RFP format .
<code>work</code>	REAL for <code>slansf</code> . DOUBLE PRECISION for <code>dlansf</code> . Workspace array, DIMENSION ($\max(1, lwork)$), where $lwork \geq n$ when <code>norm</code> = 'I' or '1' or 'O'; otherwise, <code>work</code> is not referenced.

Output Parameters

<code>val</code>	REAL for <code>slansf</code> DOUBLE PRECISION for <code>dlansf</code>
------------------	--

Value returned by the function.

?lanhf

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a Hermitian matrix in RFP format.

Syntax

```
val = clanhf(norm, transr, uplo, n, a, work)
val = zlanhf(norm, transr, uplo, n, a, work)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lanhf returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an n -by- n complex Hermitian matrix A in the [rectangular full packed \(RFP\) format](#).

Input Parameters

<i>norm</i>	CHARACTER*1. Specifies the value to be returned by the routine: = 'M' or 'm': <i>val</i> = max(abs(A_{ij})), largest absolute value of the matrix A . = '1' or 'O' or 'o': <i>val</i> = norm1(A), 1-norm of the matrix A (maximum column sum), = 'I' or 'i': <i>val</i> = normI(A), infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': <i>val</i> = normF(A), Frobenius norm of the matrix A (square root of sum of squares).
<i>transr</i>	CHARACTER*1. Specifies whether the RFP format of matrix A is normal or conjugate-transposed format. If <i>transr</i> = 'N': RFP format is normal; if <i>transr</i> = 'C': RFP format is conjugate-transposed.
<i>uplo</i>	CHARACTER*1. Specifies whether the RFP matrix A came from upper or lower triangular matrix. If <i>uplo</i> = 'U': RFP matrix A came from an upper triangular matrix; if <i>uplo</i> = 'L': RFP matrix A came from a lower triangular matrix.
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.

When $n = 0$, `?lanhf` is set to zero.

a	COMPLEX for <code>clanhf</code> DOUBLE COMPLEX for <code>zlanhf</code> Array, DIMENSION ($n*(n+1)/2$). The upper (if <code>uplo = 'U'</code>) or lower (if <code>uplo = 'L'</code>) part of the Hermitian matrix A stored in RFP format .
$work$	COMPLEX for <code>clanhf</code> . DOUBLE COMPLEX for <code>zlanhf</code> . Workspace array, DIMENSION ($\max(1, lwork)$), where $lwork \geq n$ when $norm = 'I'$ or ' 1 ' or ' O '; otherwise, $work$ is not referenced.

Output Parameters

val	COMPLEX for <code>clanhf</code> DOUBLE COMPLEX for <code>zlanhf</code> Value returned by the function.
-------	--

?tfntp

Copies a triangular matrix from the rectangular full packed format (TF) to the standard packed format (TP).

Syntax

FORTRAN 77:

```
call stfttp( transr, uplo, n, arf, ap, info )
call dtfttp( transr, uplo, n, arf, ap, info )
call ctfttp( transr, uplo, n, arf, ap, info )
call ztfttp( transr, uplo, n, arf, ap, info )
```

C:

```
lapack_int LAPACKE_<?>tfttp( int matrix_layout, char transr, char uplo, lapack_int n,
const <datatype>* arf, <datatype>* ap );
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine copies a triangular matrix A from the Rectangular Full Packed (RFP) format to the standard packed format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>transr</i>	CHARACTER*1. = 'N': <i>arf</i> is in the Normal format, = 'T': <i>arf</i> is in the Transpose format (for <code>stfttp</code> and <code>dtfttp</code>), = 'C': <i>arf</i> is in the Conjugate-transpose format (for <code>ctfttp</code> and <code>ztfttp</code>).
<i>uplo</i>	CHARACTER*1. Specifies whether <i>A</i> is upper or lower triangular: = 'U': <i>A</i> is upper triangular, = 'L': <i>A</i> is lower triangular.
<i>n</i>	INTEGER. The order of the matrix <i>A</i> . $n \geq 0$.
<i>arf</i>	REAL for <code>stfttp</code> , DOUBLE PRECISION for <code>dtfttp</code> , COMPLEX for <code>ctfttp</code> , DOUBLE COMPLEX for <code>ztfttp</code> . Array, DIMENSION at least $\max(1, n*(n+1)/2)$. On entry, the upper or lower triangular matrix <i>A</i> stored in the RFP format.

Output Parameters

<i>ap</i>	REAL for <code>stfttp</code> , DOUBLE PRECISION for <code>dtfttp</code> , COMPLEX for <code>ctfttp</code> , DOUBLE COMPLEX for <code>ztfttp</code> . Array, DIMENSION at least $\max(1, n*(n+1)/2)$. On exit, the upper or lower triangular matrix <i>A</i> , packed columnwise in a linear array. The <i>j</i> -th column of <i>A</i> is stored in the array <i>ap</i> as follows: if <i>uplo</i> = 'U', $ap(i + (j-1)*j/2) = A(i,j)$ for $1 \leq i \leq j$, if <i>uplo</i> = 'L', $ap(i + (j-1)*(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.
<i>info</i>	INTEGER. =0: successful exit, < 0: if <i>info</i> = $-i$, the <i>i</i> -th parameter had an illegal value.

?tftr

Copies a triangular matrix from the rectangular full packed format (TF) to the standard full format (TR) .

Syntax

FORTRAN 77:

```
call stfttr( transr, uplo, n, arf, a, lda, info )
call dtfttr( transr, uplo, n, arf, a, lda, info )
call cftttr( transr, uplo, n, arf, a, lda, info )
call zftttr( transr, uplo, n, arf, a, lda, info )
```

C:

```
lapack_int LAPACKE_<?>tfttr( int matrix_layout, char transr, char uplo, lapack_int n,
const <datatype>* arf, <datatype>* a, lapack_int lda );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine copies a triangular matrix A from the Rectangular Full Packed (RFP) format to the standard full format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>transr</i>	CHARACTER*1. = 'N': <i>arf</i> is in the Normal format, = 'T': <i>arf</i> is in the Transpose format (for <code>stfttr</code> and <code>dtfttr</code>), = 'C': <i>arf</i> is in the Conjugate-transpose format (for <code>cftttr</code> and <code>zftttr</code>).
<i>uplo</i>	CHARACTER*1. Specifies whether A is upper or lower triangular: = 'U': A is upper triangular, = 'L': A is lower triangular.
<i>n</i>	INTEGER. The order of the matrices <i>arf</i> and <i>a</i> . $n \geq 0$.
<i>arf</i>	REAL for <code>stfttr</code> , DOUBLE PRECISION for <code>dtfttr</code> , COMPLEX for <code>cftttr</code> , DOUBLE COMPLEX for <code>zftttr</code> . Array, DIMENSION at least $\max(1, n*(n+1)/2)$.
	On entry, the upper or lower triangular matrix A stored in the RFP format.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1,n)$.

Output Parameters

a

```
REAL for stffttr,
DOUBLE PRECISION for dtffttr,
COMPLEX for ctffttr,
DOUBLE COMPLEX for ztffttr.
```

Array, DIMENSION (*lda*, *).

On exit, the triangular matrix *A*. If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of the array *a* contains the upper triangular matrix, and the strictly lower triangular part of *a* is not referenced. If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of the array *a* contains the lower triangular matrix, and the strictly upper triangular part of *a* is not referenced.

info

```
INTEGER.  
=0: successful exit,  
< 0: if info = -i, the i-th parameter had an illegal value.
```

?tpqrt2

Computes a QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for *Q*.

Syntax

FORTRAN 77:

```
call stpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call dtpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call ctpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
call ztpqrt2(m, n, l, a, lda, b, ldb, t, ldt, info)
```

Fortran 95:

```
call tpqrt2(a, b, t [, info])
```

C:

```
lapack_int LAPACKE_stpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, float * a, lapack_int lda, float * b, lapack_int ldb, float * t, lapack_int ldt);
lapack_int LAPACKE_dtpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, double * a, lapack_int lda, double * b, lapack_int ldb, double * t, lapack_int ldt);
lapack_int LAPACKE_ctpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb, lapack_complex_float * t, lapack_int ldt);
lapack_int LAPACKE_ztpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb, lapack_complex_double * t, lapack_int ldt);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The input matrix C is an $(n+m)$ -by- n matrix

$$C = \begin{bmatrix} A \\ B \end{bmatrix} \begin{array}{l} \leftarrow n \times n \text{ upper triangular} \\ \leftarrow m \times n \text{ pentagonal} \end{array}$$

where A is an n -by- n upper triangular matrix, and B is an m -by- n pentagonal matrix consisting of an $(m-l)$ -by- n rectangular matrix $B1$ on top of an l -by- n upper trapezoidal matrix $B2$:

$$B = \begin{bmatrix} B1 \\ B2 \end{bmatrix} \begin{array}{l} \leftarrow (m-l) \times n \text{ rectangular} \\ \leftarrow l \times n \text{ upper trapezoidal} \end{array}$$

The upper trapezoidal matrix $B2$ consists of the first l rows of an n -by- n upper triangular matrix, where $0 \leq l \leq \min(m, n)$. If $l=0$, B is an m -by- n rectangular matrix. If $m=l=n$, B is upper triangular. The matrix W contains the elementary reflectors $H(i)$ in the i th column below the diagonal (of A) in the $(n+m)$ -by- n input matrix C so that W can be represented as

$$W = \begin{bmatrix} I \\ V \end{bmatrix} \begin{array}{l} \leftarrow n \times n \text{ identity} \\ \leftarrow m \times n \text{ pentagonal} \end{array}$$

Thus, V contains all of the information needed for W , and is returned in array b .

NOTE

V has the same form as B :

$$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix} \begin{array}{l} \leftarrow (m-l) \times n \text{ rectangular} \\ \leftarrow l \times n \text{ upper trapezoidal} \end{array}$$

The columns of V represent the vectors which define the $H(i)$ s.

The $(m+n)$ -by- $(m+n)$ block reflector H is then given by

$H = I - W^* T^* W^T$ for real flavors, and

$H = I - W^* T^* W^H$ for complex flavors

where W^T is the transpose of W , W^H is the conjugate transpose of W , and T is the upper triangular factor of the block reflector.

Input Parameters

m	INTEGER. The total number of rows in the matrix B ($m \geq 0$).
n	INTEGER. The number of columns in B and the order of the triangular matrix A ($n \geq 0$).
l	INTEGER. The number of rows of the upper trapezoidal part of B ($\min(m, n) \geq l \geq 0$).
a, b	REAL for <code>stpqrt2</code> DOUBLE PRECISION for <code>dtpqrt2</code> COMPLEX for <code>ctpqrt2</code> COMPLEX*16 for <code>ztpqrt2</code> . Arrays: a DIMENSION (lda, n) contains the n -by- n upper triangular matrix A . b DIMENSION (ldb, n), the pentagonal m -by- n matrix B . The first ($m-l$) rows contain the rectangular B_1 matrix, and the next l rows contain the upper trapezoidal B_2 matrix.
lda	INTEGER. The leading dimension of a ; at least $\max(1, n)$.
ldb	INTEGER. The leading dimension of b ; at least $\max(1, m)$.
ldt	INTEGER. The leading dimension of t ; at least $\max(1, n)$.

Output Parameters

a	The elements on and above the diagonal of the array contain the upper triangular matrix R .
b	The pentagonal matrix V .
t	REAL for <code>stpqrt2</code> DOUBLE PRECISION for <code>dtpqrt2</code> COMPLEX for <code>ctpqrt2</code> COMPLEX*16 for <code>ztpqrt2</code> . Array, DIMENSION (ldt, n). The upper n -by- n upper triangular factor T of the block reflector.
$info$	INTEGER. If $info = 0$, the execution is successful. If $info < 0$ and $info = -i$, the i th argument had an illegal value.

?tprfb

Applies a real or complex "triangular-pentagonal" blocked reflector to a real or complex matrix, which is composed of two blocks.

Syntax

FORTRAN 77:

```
call stprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb,
work, ldwork)

call dtprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb,
work, ldwork)

call ctprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb,
work, ldwork)

call ztprfb(side, trans, direct, storev, m, n, k, l, v, ldv, t, ldt, a, lda, b, ldb,
work, ldwork)
```

Fortran 95:

```
call tprrfb(t, v, a, b[, direct][, storev][, side][, trans])
```

C:

```
lapack_int LAPACKE_stprfb (int matrix_layout, char side, char trans, char direct, char
storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const float * v,
lapack_int ldv, const float * t, lapack_int ldt, float * a, lapack_int lda, float * b,
lapack_int ldb, lapack_int myldwork);

lapack_int LAPACKE_dtprfb (int matrix_layout, char side, char trans, char direct, char
storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const double * v,
lapack_int ldv, const double * t, lapack_int ldt, double * a, lapack_int lda, double * b,
lapack_int ldb, lapack_int myldwork);

lapack_int LAPACKE_ctprfb (int matrix_layout, char side, char trans, char direct, char
storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const
lapack_complex_float * v, lapack_int ldv, const lapack_complex_float * t, lapack_int
ldt, lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int
ldb, lapack_int myldwork);

lapack_int LAPACKE_ztprrfb (int matrix_layout, char side, char trans, char direct, char
storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const
lapack_complex_double * v, lapack_int ldv, const lapack_complex_double * t, lapack_int
ldt, lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int
ldb, lapack_int myldwork);
```

Include Files

- Fortran: mkl.fi
- Fortran 95: lapack.f90
- C: mkl.h

Description

The ?tprrfb routine applies a real or complex "triangular-pentagonal" block reflector H , H^T , or H^H from either the left or the right to a real or complex matrix C , which is composed of two blocks A and B .

The block B is m -by- n . If $side = 'R'$, A is m -by- k , and if $side = 'L'$, A is of size k -by- n .

$$\begin{array}{ll}
 \text{direct} = 'F' & \text{direct} = 'B' \\
 \text{side} = 'R' \quad C = \begin{bmatrix} A & B \end{bmatrix} & C = \begin{bmatrix} B & A \end{bmatrix} \\
 \text{side} = 'L' \quad C = \begin{bmatrix} A \\ B \end{bmatrix} & C = \begin{bmatrix} B \\ A \end{bmatrix}
 \end{array}$$

The pentagonal matrix V is composed of a rectangular block $V1$ and a trapezoidal block $V2$. The size of the trapezoidal block is determined by the parameter ℓ , where $0 \leq \ell \leq k$. If $\ell=k$, the $V2$ block of V is triangular; if $\ell=0$, there is no trapezoidal block, thus $V = V1$ is rectangular.

<code>storev='C'</code>	$V = \begin{bmatrix} V1 \\ V2 \end{bmatrix}$	$V = \begin{bmatrix} V2 \\ V1 \end{bmatrix}$
	$V2$ is upper trapezoidal (first ℓ rows of k -by- k upper triangular)	$V2$ is lower trapezoidal (last ℓ rows of k -by- k lower triangular matrix)
<code>storev='R'</code>	$V = \begin{bmatrix} V1 & V2 \end{bmatrix}$	$V = \begin{bmatrix} V2 & V1 \end{bmatrix}$
	$V2$ is lower trapezoidal (first ℓ columns of k -by- k lower triangular matrix)	$V2$ is upper trapezoidal (last ℓ columns of k -by- k upper triangular matrix)
<code>storev='C'</code>	V is m -by- k $V2$ is ℓ -by- k	V is n -by- k $V2$ is ℓ -by- k
<code>storev='R'</code>	V is k -by- m $V2$ is k -by- ℓ	V is k -by- n $V2$ is k -by- ℓ

Input Parameters

<code>side</code>	CHARACTER*1. = 'L': apply H , H^T , or H^H from the left, = 'R': apply H , H^T , or H^H from the right.
<code>trans</code>	CHARACTER*1. = 'N': apply H (no transpose), = 'T': apply H^T (transpose), = 'C': apply H^H (conjugate transpose).
<code>direct</code>	CHARACTER*1. Indicates how H is formed from a product of elementary reflectors:

= 'F': $H = H(1) H(2) \dots H(k)$ (Forward),
 = 'B': $H = H(k) \dots H(2) H(1)$ (Backward).

storev

CHARACTER*1.

Indicates how the vectors that define the elementary reflectors are stored:

= 'C': Columns,
 = 'R': Rows.

*m*INTEGER. The total number of rows in the matrix B ($m \geq 0$).*n*INTEGER. The number of columns in B ($n \geq 0$).*k*INTEGER. The order of the matrix T , which is the number of elementary reflectors whose product defines the block reflector. ($k \geq 0$)*l*INTEGER. The order of the trapezoidal part of V . ($k \geq l \geq 0$).*v*

REAL for stprfb

DOUBLE PRECISION for dtprfb

COMPLEX for ctprfb

COMPLEX*16 for ztprfb.

DIMENSION (ldv, k) if *storev* = 'C',DIMENSION (ldv, m) if *storev* = 'R' and *side* = 'L',DIMENSION (ldv, n) if *storev* = 'R' and *side* = 'R'.The pentagonal matrix V , which contains the elementary reflectors $H(1)$, $H(2)$, ..., $H(k)$.*ldv*INTEGER. The leading dimension of the array *v*.If *storev* = 'C' and *side* = 'L', at least max(1, *m*).If *storev* = 'C' and *side* = 'R', at least max(1, *n*).If *storev* = 'R', at least *k*.*t*

REAL for stprfb

DOUBLE PRECISION for dtprfb

COMPLEX for ctprfb

COMPLEX*16 for ztprfb.

DIMENSION (ldt, k). The triangular k -by- k matrix T in the representation of the block reflector.*ldt*INTEGER. The leading dimension of the array *t* ($ldt \geq k$).*a*

REAL for stprfb

DOUBLE PRECISION for dtprfb

COMPLEX for ctprfb

COMPLEX*16 for ztprfb.

DIMENSION (*lda*, *n*) if *side* = 'L',
 DIMENSION (*lda*, *k*) if *side* = 'R'.
 The *k*-by-*n* or *m*-by-*k* matrix *A*.

lda INTEGER. The leading dimension of the array *a*.
 If *side* = 'L', at least max(1, *k*).
 If *side* = 'R', at least max(1, *m*).

b REAL for stprfb
 DOUBLE PRECISION for dtprfb
 COMPLEX for ctprfb
 COMPLEX*16 for ztprfb.
 DIMENSION (*ldb*, *n*), the *m*-by-*n* matrix *B*.

ldb INTEGER. The leading dimension of the array *b* (*ldb* $\geq \max(1, m)$).

work REAL for stprfb
 DOUBLE PRECISION for dtprfb
 COMPLEX for ctprfb
 COMPLEX*16 for ztprfb.
 DIMENSION (*ldwork*, *n*) if *side* = 'L',
 DIMENSION (*ldwork*, *k*) if *side* = 'R'.
 Workspace array.

ldwork INTEGER. The leading dimension of the array *work*.
 If *side* = 'L', at least *k*.
 If *side* = 'R', at least *m*.

Output Parameters

a Contains the corresponding block of $H*C$, H^T*C , H^H*C , $C*H$, $C*H^T$, or $C*H^H$.
b Contains the corresponding block of $H*C$, H^T*C , H^H*C , $C*H$, $C*H^T$, or $C*H^H$.

?tpdff

Copies a triangular matrix from the standard packed format (TP) to the rectangular full packed format (TF).

Syntax

FORTRAN 77:

```
call stpttf( transr, uplo, n, ap, arf, info )
call dtpttf( transr, uplo, n, ap, arf, info )
call ctpttf( transr, uplo, n, ap, arf, info )
call ztpttf( transr, uplo, n, ap, arf, info )
```

C:

```
lapack_int LAPACKE_<?>tpttf( int matrix_layout, char transr, char uplo, lapack_int n,  
const <datatype>* ap, <datatype>* arf );
```

Include Files

- Fortran: mkl.fi
 - C: mkl.h

Description

The routine copies a triangular matrix A from the standard packed format to the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

The data types are given for the Fortran interface. A `<datatype>` placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

transr CHARACTER*1.
 = 'N': *arf* must be in the Normal format,
 = 'T': *arf* must be in the Transpose format (for *stpttf* and *dtpttf*),
 = 'C': *arf* must be in the Conjugate-transpose format (for *ctpttf* and
 ztpttf).

uplo CHARACTER*1.
Specifies whether A is upper or lower triangular:
= 'U': A is upper triangular,
= 'L': A is lower triangular.

n INTEGER; The order of the matrix A ; $n \geq 0$.

ap REAL for stpttf,
 DOUBLE PRECISION for dtpttf,
 COMPLEX for ctpttf,
 DOUBLE COMPLEX for ztpttf.
Array, DIMENSION at least $\max(1, n*(n+1)/2)$.

On entry, the upper or lower triangular matrix A , packed columnwise in a linear array. The j -th column of A is stored in the array ap as follows:

if $uplo = 'U'$, $ap(i + (j-1)*j/2) = A(i,j)$ for $1 \leq i \leq j$,
 if $uplo = 'L'$, $ap(i + (j-1)*(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

Output Parameters

DOUBLE COMPLEX for `ztpttf`.
Array, DIMENSION at least $\max(1, n*(n+1)/2)$.
 On exit, the upper or lower triangular matrix A stored in the RFP format.

info INTEGER.
 =0: successful exit,
 < 0: if *info* = -*i*, the *i*-th parameter had an illegal value.

?tptr

Copies a triangular matrix from the standard packed format (TP) to the standard full format (TR).

Syntax

FORTRAN 77:

```
call stptr( uplo, n, ap, a, lda, info )
call dtptr( uplo, n, ap, a, lda, info )
call cptptr( uplo, n, ap, a, lda, info )
call ztptr( uplo, n, ap, a, lda, info )
```

C:

```
lapack_int LAPACKE_<?>tptr( int matrix_layout, char uplo, lapack_int n,
const <datatype>* ap, <datatype>* a, lapack_int lda );
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine copies a triangular matrix A from the standard packed format to the standard full format.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo CHARACTER*1.
 Specifies whether A is upper or lower triangular:
 = 'U': A is upper triangular,
 = 'L': A is lower triangular.

n INTEGER. The order of the matrices *ap* and *a*. $n \geq 0$.

ap REAL for `stptr`,
 DOUBLE PRECISION for `dtptr`,

COMPLEX for `ctpttr`,
 DOUBLE COMPLEX for `ztpttr`.
 Array, DIMENSION at least $\max(1, n*(n+1)/2)$.
 On entry, the upper or lower triangular matrix A , packed columnwise in a linear array. The j -th column of A is stored in the array `ap` as follows:
 if $uplo = 'U'$, $ap(i + (j-1)*j/2) = A(i,j)$ for $1 \leq i \leq j$,
 if $uplo = 'L'$, $ap(i + (j-1)*(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.
`lda` INTEGER. The leading dimension of the array `a`. $lda \geq \max(1,n)$.

Output Parameters

`a` REAL for `stpttr`,
 DOUBLE PRECISION for `dtpttr`,
 COMPLEX for `ctpttr`,
 DOUBLE COMPLEX for `ztpttr`.
 Array, DIMENSION (`lda, *`).
 On exit, the triangular matrix A . If $uplo = 'U'$, the leading n -by- n upper triangular part of the array `a` contains the upper triangular part of the matrix A , and the strictly lower triangular part of `a` is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of the array `a` contains the lower triangular part of the matrix A , and the strictly upper triangular part of `a` is not referenced.
`info` INTEGER.
 =0: successful exit,
 < 0: if $info = -i$, the i -th parameter had an illegal value.

?trttf

Copies a triangular matrix from the standard full format (TR) to the rectangular full packed format (TF).

Syntax

FORTRAN 77:

```
call strttf( transr, uplo, n, a, lda, arf, info )
call dtrttf( transr, uplo, n, a, lda, arf, info )
call ctrttf( transr, uplo, n, a, lda, arf, info )
call ztrttf( transr, uplo, n, a, lda, arf, info )
```

C:

```
lapack_int LAPACKE_<?>trttf( int matrix_layout, char transr, char uplo, lapack_int n,
const <datatype>* a, lapack_int lda, <datatype>* arf );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine copies a triangular matrix A from the standard full format to the Rectangular Full Packed (RFP) format. For the description of the RFP format, see [Matrix Storage Schemes](#).

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

<i>transr</i>	CHARACTER*1. = 'N': <i>arf</i> must be in the Normal format, = 'T': <i>arf</i> must be in the Transpose format (for strttf and dtrttf), = 'C': <i>arf</i> must be in the Conjugate-transpose format (for ctrttf and ztrttf).
<i>uplo</i>	CHARACTER*1. Specifies whether A is upper or lower triangular: = 'U': A is upper triangular, = 'L': A is lower triangular.
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>a</i>	REAL for strttf, DOUBLE PRECISION for dtrttf, COMPLEX for ctrttf, DOUBLE COMPLEX for ztrttf. <i>Array</i> , DIMENSION (<i>lda</i> , *). On entry, the triangular matrix A . If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of the array <i>a</i> contains the upper triangular matrix, and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of the array <i>a</i> contains the lower triangular matrix, and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1,n)$.

Output Parameters

<i>arf</i>	REAL for strttf, DOUBLE PRECISION for dtrttf, COMPLEX for ctrttf, DOUBLE COMPLEX for ztrttf. <i>Array</i> , DIMENSION at least $\max(1, n*(n+1)/2)$.
------------	---

On exit, the upper or lower triangular matrix A stored in the RFP format.

info INTEGER.
 $=0$: successful exit,
 < 0 : if $info = -i$, the i -th parameter had an illegal value.

?trttp

Copies a triangular matrix from the standard full format (TR) to the standard packed format (TP).

Syntax

FORTRAN 77:

```
call strttp( uplo, n, a, lda, ap, info )
call dtrttp( uplo, n, a, lda, ap, info )
call ctrttp( uplo, n, a, lda, ap, info )
call ztrttp( uplo, n, a, lda, ap, info )
```

C:

```
lapack_int LAPACKE_<?>trttp( int matrix_layout, char uplo, lapack_int n,
const <datatype>* a, lapack_int lda, <datatype>* ap );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine copies a triangular matrix A from the standard full format to the standard packed format.

Input Parameters

The data types are given for the Fortran interface. A <datatype> placeholder, if present, is used for the C interface data types in the C interface section above. See the [C Interface Conventions](#) section for the C interface principal conventions and type definitions.

uplo CHARACTER*1.
 Specifies whether A is upper or lower triangular:
 $= 'U'$: A is upper triangular,
 $= 'L'$: A is lower triangular.

n INTEGER. The order of the matrices a and ap . $n \geq 0$.

a REAL for strttp,
 DOUBLE PRECISION for dtrttp,
 COMPLEX for ctrttp,
 DOUBLE COMPLEX for ztrttp.
 Array, DIMENSION (lda, n).

On entry, the triangular matrix A . If $\text{uplo} = \text{'U'}$, the leading n -by- n upper triangular part of the array a contains the upper triangular matrix, and the strictly lower triangular part of a is not referenced. If $\text{uplo} = \text{'L'}$, the leading n -by- n lower triangular part of the array a contains the lower triangular matrix, and the strictly upper triangular part of a is not referenced.

lda INTEGER. The leading dimension of the array a . $\text{lda} \geq \max(1,n)$.

Output Parameters

ap REAL for strttp ,
 DOUBLE PRECISION for dtrttp ,
 COMPLEX for ctrttp ,
 DOUBLE COMPLEX for ztrttp .
 Array, DIMENSION at least $\max(1, n*(n+1)/2)$.
 On exit, the upper or lower triangular matrix A , packed columnwise in a linear array. The j -th column of A is stored in the array ap as follows:
 if $\text{uplo} = \text{'U'}$, $ap(i + (j-1)*j/2) = A(i,j)$ for $1 \leq i \leq j$,
 if $\text{uplo} = \text{'L'}$, $ap(i + (j-1)*(2n-j)/2) = A(i,j)$ for $j \leq i \leq n$.

$info$ INTEGER.
 =0: successful exit,
 < 0: if $info = -i$, the i -th parameter had an illegal value.

?pstf2

Computes the Cholesky factorization with complete pivoting of a real symmetric or complex Hermitian positive semi-definite matrix.

Syntax

```
call spstf2( uplo, n, a, lda, piv, rank, tol, work, info )
call dpstf2( uplo, n, a, lda, piv, rank, tol, work, info )
call cpstf2( uplo, n, a, lda, piv, rank, tol, work, info )
call zpstf2( uplo, n, a, lda, piv, rank, tol, work, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The real flavors `spstf2` and `dpstf2` compute the Cholesky factorization with complete pivoting of a real symmetric positive semi-definite matrix A . The complex flavors `cpstf2` and `zpstf2` compute the Cholesky factorization with complete pivoting of a complex Hermitian positive semi-definite matrix A . The factorization has the form:

$$P^T * A * P = U^T * U, \text{ if } \text{uplo} = \text{'U'} \text{ for real flavors,}$$

$P^T * A * P = U^H * U$, if $\text{uplo} = \text{'U'}$ for complex flavors,

$P^T * A * P = L * L^T$, if $\text{uplo} = \text{'L'}$ for real flavors,

$P^T * A * P = L * L^H$, if $\text{uplo} = \text{'L'}$ for complex flavors,

where U is an upper triangular matrix and L is lower triangular, and P is stored as vector piv .

This algorithm does not check that A is positive semi-definite. This version of the algorithm calls [level 2 BLAS](#).

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the symmetric or Hermitian matrix A is stored:

= 'U': Upper triangular,

= 'L': Lower triangular.

n

INTEGER. The order of the matrix A . $n \geq 0$.

a

REAL for spstf2 ,
 DOUBLE PRECISION for dpstf2 ,
 COMPLEX for cpstf2 ,
 DOUBLE COMPLEX for zpstf2 .

Array, DIMENSION ($\text{lda}, *$).

On entry, the symmetric matrix A . If $\text{uplo} = \text{'U'}$, the leading n -by- n upper triangular part of the array a contains the upper triangular part of the matrix A , and the strictly lower triangular part of a is not referenced. If $\text{uplo} = \text{'L'}$, the leading n -by- n lower triangular part of the array a contains the lower triangular part of the matrix A , and the strictly upper triangular part of a is not referenced.

tol

REAL for spstf2 and cpstf2 ,
 DOUBLE PRECISION for dpstf2 and zpstf2 .

A user-defined tolerance.

If $\text{tol} < 0$, $n * \text{ulp} * \max(A(k,k))$ will be used (ulp is the Unit in the Last Place, or Unit of Least Precision). The algorithm terminates at the $(k - 1)$ -st step if the pivot is not greater than tol .

lda

INTEGER. The leading dimension of the matrix A . $\text{lda} \geq \max(1,n)$.

work

REAL for spstf2 and cpstf2 ,
 DOUBLE PRECISION for dpstf2 and zpstf2 .
 Workspace array, DIMENSION at least $\max(1, 2*n)$.

Output Parameters

piv

INTEGER. Array. DIMENSION at least $\max(1,n)$.

piv is such that the non-zero entries are $P(\text{piv}(k), k) = 1$.

<i>a</i>	On exit, if <i>info</i> = 0, the factor U or L from the Cholesky factorization stored the same way as the matrix <i>A</i> is stored on entry.
<i>rank</i>	INTEGER. The rank of <i>A</i> , determined by the number of steps the algorithm completed.
<i>info</i>	INTEGER. < 0: if <i>info</i> = - <i>k</i> , the <i>k</i> -th parameter had an illegal value, =0: the algorithm completed successfully, > 0: the matrix <i>A</i> is rank-deficient with the computed rank, returned in <i>rank</i> , or indefinite.

dlat2s

Converts a double-precision triangular matrix to a single-precision triangular matrix.

Syntax

```
call dlat2s( uplo, n, a, lda, sa, ldsa, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This routine converts a double-precision triangular matrix *A* to a single-precision triangular matrix *SA*. dlat2s checks that all the elements of *A* are between -*RMAX* and *RMAX*, where *RMAX* is the overflow for the single-precision arithmetic. If this condition is not met, the conversion is aborted and a flag is raised. The routine does no parameter checking.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular: = 'U': <i>A</i> is upper triangular, = 'L': <i>A</i> is lower triangular.
<i>n</i>	INTEGER. The number of rows and columns of the matrix <i>A</i> . <i>n</i> ≥ 0.
<i>a</i>	DOUBLE PRECISION. Array, DIMENSION (<i>lda</i> , *). On entry, the <i>n</i> -by- <i>n</i> triangular matrix <i>A</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . <i>lda</i> ≥ <i>max</i> (1, <i>n</i>).
<i>ldsa</i>	INTEGER. The leading dimension of the array <i>sa</i> . <i>ldsa</i> ≥ <i>max</i> (1, <i>n</i>).

Output Parameters

<i>sa</i>	REAL. Array, DIMENSION (<i>ldsa</i> , *). Only the part of <i>sa</i> determined by <i>uplo</i> is referenced. On exit,
	<ul style="list-style-type: none"> • if <i>info</i> = 0, the <i>n</i>-by-<i>n</i> triangular matrix <i>SA</i>, • if <i>info</i> > 0, the content of the part of <i>sa</i> determined by <i>uplo</i> is unspecified.

<i>info</i>	INTEGER. =0: successful exit, > 0: an element of the matrix <i>A</i> is greater than the single-precision overflow threshold; in this case, the content of the part of <i>sa</i> determined by <i>uplo</i> is unspecified on exit.
-------------	--

zlat2c

Converts a double complex triangular matrix to a complex triangular matrix.

Syntax

```
call zlat2c( uplo, n, a, lda, sa, ldsa, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This routine is declared in mkl_lapack.fi for FORTRAN 77 interface and in mkl_lapack.h for C interface.

The routine converts a DOUBLE COMPLEX triangular matrix *A* to a COMPLEX triangular matrix *SA*. zlat2c checks that the real and complex parts of all the elements of *A* are between -RMAX and RMAX, where RMAX is the overflow for the single-precision arithmetic. If this condition is not met, the conversion is aborted and a flag is raised. The routine does no parameter checking.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix <i>A</i> is upper or lower triangular: = 'U': <i>A</i> is upper triangular, = 'L': <i>A</i> is lower triangular.
<i>n</i>	INTEGER. The number of rows and columns in the matrix <i>A</i> . <i>n</i> ≥ 0.
<i>a</i>	DOUBLE COMPLEX. Array, DIMENSION (<i>lda</i> , *). On entry, the <i>n</i> -by- <i>n</i> triangular matrix <i>A</i> .

lda INTEGER. The leading dimension of the array *a*. $lda \geq max(1,n)$.

ldsa INTEGER. The leading dimension of the array *sa*. $ldsa \geq max(1,n)$.

Output Parameters

sa COMPLEX.

Array, DIMENSION (*ldsa, **).

Only the part of *sa* determined by *uplo* is referenced. On exit,

- if *info* = 0, the *n*-by-*n* triangular matrix *sa*,
- if *info* > 0, the content of the part of *sa* determined by *uplo* is unspecified.

info INTEGER.

=0: successful exit,
 > 0: the real or complex part of an element of the matrix *A* is greater than the single-precision overflow threshold; in this case, the content of the part of *sa* determined by *uplo* is unspecified on exit.

?lacp2

Copies all or part of a real two-dimensional array to a complex array.

Syntax

FORTRAN 77:

```
call clacp2( uplo, m, n, a, lda, b, ldb )
call zlacp2( uplo, m, n, a, lda, b, ldb )
```

C:

```
lapack_int LAPACKE_clacp2 (int matrix_layout , char uplo , lapack_int m , lapack_int n , const float * a , lapack_int lda , lapack_complex_float * b , lapack_int ldb );
lapack_int LAPACKE_zlacp2 (int matrix_layout , char uplo , lapack_int m , lapack_int n , const double * a , lapack_int lda , lapack_complex_double * b , lapack_int ldb );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine copies all or part of a two-dimensional matrix *A* to another matrix *B*.

Input Parameters

<i>matrix_layout</i>	Specifies whether matrix storage layout is row major (<code>LAPACK_ROW_MAJOR</code>) or column major (<code>LAPACK_COL_MAJOR</code>).
<i>uplo</i>	CHARACTER*1.

Specifies the part of the matrix A to be copied to B .

If $uplo = 'U'$, the upper triangular part of A ;

if $uplo = 'L'$, the lower triangular part of A .

Otherwise, all of the matrix A is copied.

m INTEGER. The number of rows in the matrix A ($m \geq 0$).

n INTEGER. The number of columns in A ($n \geq 0$).

a REAL for clacp2

DOUBLE PRECISION for zlacp2

Array $a(1da, n)$, contains the m -by- n matrix A .

If $uplo = 'U'$, only the upper triangle or trapezoid is accessed; if $uplo = 'L'$, only the lower triangle or trapezoid is accessed.

$1da$ INTEGER. The leading dimension of a ; $1da \geq \max(1, m)$.

ldb INTEGER. The leading dimension of the output array b ; $ldb \geq \max(1, m)$.

Output Parameters

b COMPLEX for clacp2

DOUBLE COMPLEX for zlacp2.

Array $b(ldb, m)$, contains the m -by- n matrix B .

On exit, $B = A$ in the locations specified by $uplo$.

?la_gbamv

Performs a matrix-vector operation to calculate error bounds.

Syntax

FORTRAN 77:

```
call sla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
call dla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
call cla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
call zla_gbamv(trans, m, n, kl, ku, alpha, ab, ldab, x, incx, beta, y, incy)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?la_gbamv function performs one of the matrix-vector operations defined as

$y := \alpha * \text{abs}(A) * \text{abs}(x) + \beta * \text{abs}(y)$,

or

$y := \alpha \cdot \text{abs}(A)^T \cdot \text{abs}(x) + \beta \cdot \text{abs}(y),$

where:

α and β are scalars,

x and y are vectors,

A is an m -by- n matrix, with k_l sub-diagonals and k_u super-diagonals.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by $(n + 1)$ times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

<i>trans</i>	INTEGER. Specifies the operation to be performed: If <i>trans</i> = 'BLAS_NO_TRANS', then $y := \alpha \cdot \text{abs}(A) \cdot \text{abs}(x) + \beta \cdot \text{abs}(y)$ If <i>trans</i> = 'BLAS_TRANS', then $y := \alpha \cdot \text{abs}(A^T) \cdot \text{abs}(x) + \beta \cdot \text{abs}(y)$ If <i>trans</i> = 'BLAS_CONJ_TRANS', then $y := \alpha \cdot \text{abs}(A^T) \cdot \text{abs}(x) + \beta \cdot \text{abs}(y)$ The parameter is unchanged on exit.
<i>m</i>	INTEGER. Specifies the number of rows of the matrix A . The value of <i>m</i> must be at least zero. Unchanged on exit.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix A . The value of <i>n</i> must be at least zero. Unchanged on exit.
<i>k_l</i>	INTEGER. Specifies the number of sub-diagonals within the band of A . $k_l \geq 0$.
<i>k_u</i>	INTEGER. Specifies the number of super-diagonals within the band of A . $k_u \geq 0$.
<i>alpha</i>	REAL for <code>sla_gbamv</code> and <code>cla_gbamv</code> DOUBLE PRECISION for <code>dla_gbamv</code> and <code>zla_gbamv</code> Specifies the scalar α . Unchanges on exit.
<i>ab</i>	REAL for <code>sla_gbamv</code> DOUBLE PRECISION for <code>dla_gbamv</code> COMPLEX for <code>cla_gbamv</code> DOUBLE COMPLEX for <code>zla_gbamv</code> <code>Array, DIMENSION (1dab, *)</code> . Before entry, the leading m -by- n part of the array <i>ab</i> must contain the matrix of coefficients. The second dimension of <i>ab</i> must be at least $\max(1, n)$. Unchanged on exit.

<i>ldab</i>	INTEGER. Specifies the leading dimension of <i>ab</i> as declared in the calling (sub)program. The value of <i>ldab</i> must be at least $\max(1, m)$. Unchanged on exit.
<i>x</i>	<p>REAL for <i>sla_gbamv</i></p> <p>DOUBLE PRECISION for <i>dla_gbamv</i></p> <p>COMPLEX for <i>cla_gbamv</i></p> <p>DOUBLE COMPLEX for <i>zla_gbamv</i></p> <p>Array, DIMENSION $(1 + (n - 1) * \text{abs}(incx))$ when <i>trans</i> = 'N' or 'n' and at least $(1 + (m - 1) * \text{abs}(incx))$ otherwise.</p> <p>Before entry, the incremented array <i>x</i> must contain the vector <i>x</i>.</p>
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . <i>incx</i> must not be zero.
<i>beta</i>	<p>REAL for <i>sla_gbamv</i> and <i>cla_gbamv</i></p> <p>DOUBLE PRECISION for <i>dla_gbamv</i> and <i>zla_gbamv</i></p> <p>Specifies the scalar <i>beta</i>. When <i>beta</i> is zero, you do not need to set <i>y</i> on input.</p>
<i>y</i>	<p>REAL for <i>sla_gbamv</i> and <i>cla_gbamv</i></p> <p>DOUBLE PRECISION for <i>dla_gbamv</i> and <i>zla_gbamv</i></p> <p>Array, DIMENSION at least $(1 + (m - 1) * \text{abs}(incy))$ when <i>trans</i> = 'N' or 'n' and at least $(1 + (n - 1) * \text{abs}(incy))$ otherwise.</p> <p>Before entry with <i>beta</i> non-zero, the incremented array <i>y</i> must contain the vector <i>y</i>.</p>
<i>incy</i>	<p>INTEGER. Specifies the increment for the elements of <i>y</i>.</p> <p>The value of <i>incy</i> must not be zero. Unchanged on exit.</p>

Output Parameters

<i>y</i>	Updated vector <i>y</i> .
----------	---------------------------

?la_gbrcond

Estimates the Skeel condition number for a general banded matrix.

Syntax

FORTRAN 77:

```
call sla_gbrcond( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, cmode, c, info, work,
iwork )

call dla_gbrcond( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, cmode, c, info, work,
iwork )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function estimates the Skeel condition number of

$$\text{op}(A) * \text{op}2(C)$$

where

the *cmode* parameter determines *op2* as follows:

<i>cmode</i> Value	op2(C)
1	C
0	I
-1	$\text{inv}(C)$

The Skeel condition number

$$\text{cond}(A) = \text{norminf}(|\text{inv}(A)| |A|)$$

is computed by computing scaling factors R such that

$$\text{diag}(R) * A * \text{op}2(C)$$

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Specifies the form of the system of equations: If <i>trans</i> = 'N', the system has the form $A*X = B$. If <i>trans</i> = 'T', the system has the form $A^T*X = B$. If <i>trans</i> = 'C', the system has the form $A^H*X = B$.
<i>n</i>	INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.
<i>kl</i>	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
<i>ab</i> , <i>afb</i> , <i>c</i> , <i>work</i>	REAL for <i>sla_gbrcond</i> DOUBLE PRECISION for <i>dla_gbrcond</i> Arrays: <i>ab</i> (<i>ldab</i> ,*) contains the original band matrix A stored in rows from 1 to $kl + ku + 1$. The j -th column of A is stored in the j -th column of the array <i>ab</i> as follows:

```

 $ab(ku+1+i-j, j) = A(i, j)$ 
for
 $\max(1, j-ku) \leq i \leq \min(n, j+k1)$ 

 $afb(1:ldafb, *)$  contains details of the LU factorization of the band matrix  $A$ , as
returned by  $?gbtrf$ .  $U$  is stored as an upper triangular band matrix with  $k1+ku$ 
superdiagonals in rows 1 to  $k1+ku+1$ , and the multipliers used during the
factorization are stored in rows  $k1+ku+2$  to  $2*k1+ku+1$ .

 $c$ , DIMENSION  $n$ . The vector  $C$  in the formula  $\text{op}(A) * \text{op2}(C)$ .
 $work$  is a workspace array of DIMENSION  $(5*n)$ .
The second dimension of  $ab$  and  $afb$  must be at least  $\max(1, n)$ .

 $ldab$  INTEGER. The leading dimension of the array  $ab$ .  $ldab \geq k1+ku+1$ .
 $ldafb$  INTEGER. The leading dimension of  $afb$ .  $ldafb \geq 2*k1+ku+1$ .
 $ipiv$  INTEGER.
Array with DIMENSION  $n$ . The pivot indices from the factorization  $A = P*L*U$  as
computed by  $?gbtrf$ . Row  $i$  of the matrix was interchanged with row  $ipiv(i)$ .
 $cmode$  INTEGER. Determines  $\text{op2}(C)$  in the formula  $\text{op}(A) * \text{op2}(C)$  as follows:
If  $cmode = 1$ ,  $\text{op2}(C) = C$ .
If  $cmode = 0$ ,  $\text{op2}(C) = I$ .
If  $cmode = -1$ ,  $\text{op2}(C) = \text{inv}(C)$ .
 $iwork$  INTEGER. Workspace array with DIMENSION  $n$ .
```

Output Parameters

$info$ INTEGER.
If $info = 0$, the execution is successful.
If $i > 0$, the i -th parameter is invalid.

See Also

$?gbtrf$

[?la_gbrcond_c](#)

Computes the infinity norm condition number of
 $\text{op}(A)*\text{inv}(\text{diag}(c))$ for general banded matrices.

Syntax

FORTRAN 77:

```

call cla_gbrcond_c( trans, n, k1, ku, ab, ldab, afb, ldafb, ipiv, c, capply, info,
work, rwork )

call zla_gbrcond_c( trans, n, k1, ku, ab, ldab, afb, ldafb, ipiv, c, capply, info,
work, rwork )

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function computes the infinity norm condition number of

```
op(A) * inv(diag(c))
```

where the *c* is a REAL vector for cla_gbrcond_c and a DOUBLE PRECISION vector for zla_gbrcond_c.

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Specifies the form of the system of equations: If <i>trans</i> = 'N', the system has the form $A^*X = B$ (No transpose) If <i>trans</i> = 'T', the system has the form $A^T*X = B$ (Transpose) If <i>trans</i> = 'C', the system has the form $A^{H*}X = B$ (Conjugate Transpose = Transpose)
<i>n</i>	INTEGER. The number of linear equations, that is, the order of the matrix <i>A</i> ; <i>n</i> ≥ 0 .
<i>kl</i>	INTEGER. The number of subdiagonals within the band of <i>A</i> ; <i>kl</i> ≥ 0 .
<i>ku</i>	INTEGER. The number of superdiagonals within the band of <i>A</i> ; <i>ku</i> ≥ 0 .
<i>ab</i> , <i>afb</i> , <i>work</i>	COMPLEX for cla_gbrcond_c DOUBLE COMPLEX for zla_gbrcond_c Arrays: <i>ab</i> (<i>ldab</i> ,*) contains the original band matrix <i>A</i> stored in rows from 1 to <i>kl</i> + <i>ku</i> + 1. The <i>j</i> -th column of <i>A</i> is stored in the <i>j</i> -th column of the array <i>ab</i> as follows: <i>ab</i> (<i>ku</i> +1+ <i>i</i> - <i>j</i> , <i>j</i>) = <i>A</i> (<i>i</i> , <i>j</i>) for <i>max</i> (1, <i>j</i> - <i>ku</i>) $\leq i \leq \min(n, j+k1)$ <i>afb</i> (<i>ldafb</i> ,*) contains details of the LU factorization of the band matrix <i>A</i> , as returned by ?gbtrf. <i>U</i> is stored as an upper triangular band matrix with <i>kl+ku</i> superdiagonals in rows 1 to <i>kl+ku</i> +1, and the multipliers used during the factorization are stored in rows <i>kl+ku</i> +2 to 2*k _{l+ku} +1. <i>work</i> is a workspace array of DIMENSION (5*n). The second dimension of <i>ab</i> and <i>afb</i> must be at least <i>max</i> (1, <i>n</i>). <i>ldab</i> INTEGER. The leading dimension of the array <i>ab</i> . <i>ldab</i> $\geq k1+ku+1$. <i>ldafb</i> INTEGER. The leading dimension of <i>afb</i> . <i>ldafb</i> $\geq 2*kl+ku+1$. <i>ipiv</i> INTEGER.

Array with DIMENSION n . The pivot indices from the factorization $A = P^* L^* U$ as computed by `?gbtrf`. Row i of the matrix was interchanged with row $ipiv(i)$.

$c, rwork$	REAL for <code>cla_gbrcond_c</code> DOUBLE PRECISION for <code>zla_gbrcond_c</code> Array c with DIMENSION n . The vector c in the formula $op(A) * \text{inv}(\text{diag}(c))$. Array $rwork$ with DIMENSION n is a workspace.
$capply$	LOGICAL. If .TRUE., then the function uses the vector c from the formula $op(A) * \text{inv}(\text{diag}(c))$.

Output Parameters

$info$	INTEGER. If $info = 0$, the execution is successful. If $i > 0$, the i -th parameter is invalid.
--------	--

See Also

`?gbtrf`

`?la_gbrcond_x`

Computes the infinity norm condition number of
 $op(A)^*\text{diag}(x)$ for general banded matrices.

Syntax

FORTRAN 77:

```
call cla_gbrcond_x( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, x, info, work,
rwork )

call zla_gbrcond_x( trans, n, kl, ku, ab, ldab, afb, ldafb, ipiv, x, info, work,
rwork )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The function computes the infinity norm condition number of

$op(A) * \text{diag}(x)$

where the x is a COMPLEX vector for `cla_gbrcond_x` and a DOUBLE COMPLEX vector for `zla_gbrcond_x`.

Input Parameters

$trans$	CHARACTER*1. Must be 'N' or 'T' or 'C'. Specifies the form of the system of equations: If $trans = 'N'$, the system has the form $A*X = B$ (No transpose)
---------	--

If $\text{trans} = \text{'T'}$, the system has the form $A^T X = B$ (Transpose)

If $\text{trans} = \text{'C'}$, the system has the form $A^H X = B$ (Conjugate Transpose = Transpose)

n INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.

k1 INTEGER. The number of subdiagonals within the band of A ; $k1 \geq 0$.

ku INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.

ab, afb, x, work COMPLEX for `cla_gbrcond_x`

DOUBLE COMPLEX for `zla_gbrcond_x`

Arrays:

ab(1:dab,)* contains the original band matrix A stored in rows from 1 to $k1 + ku + 1$. The j -th column of A is stored in the j -th column of the array *ab* as follows:

ab(ku+1+i-j, j) = A(i, j)

for

$\max(1, j-ku) \leq i \leq \min(n, j+k1)$

afb(1:dafb,)* contains details of the LU factorization of the band matrix A , as returned by `?gbtrf`. U is stored as an upper triangular band matrix with $k1+ku$ superdiagonals in rows 1 to $k1+ku+1$, and the multipliers used during the factorization are stored in rows $k1+ku+2$ to $2*k1+ku+1$.

x, DIMENSION n . The vector *x* in the formula $\text{op}(A) * \text{diag}(x)$.

work is a workspace array of DIMENSION $(2*n)$.

The second dimension of *ab* and *afb* must be at least $\max(1, n)$.

ldab INTEGER. The leading dimension of the array *ab*. $ldab \geq k1+ku+1$.

ldafb INTEGER. The leading dimension of *afb*. $ldafb \geq 2*k1+ku+1$.

ipiv INTEGER.

Array with DIMENSION n . The pivot indices from the factorization $A = P^* L^* U$ as computed by `?gbtrf`. Row i of the matrix was interchanged with row *ipiv(i)*.

rwork REAL for `cla_gbrcond_x`

DOUBLE PRECISION for `zla_gbrcond_x`

Array *rwork* with DIMENSION n is a workspace.

Output Parameters

info INTEGER.

If $\text{info} = 0$, the execution is successful.

If $i > 0$, the i -th parameter is invalid.

See Also

[?gbtrf](#)

?la_gbrfsx_extended

Improves the computed solution to a system of linear equations for general banded matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

FORTRAN 77:

```
call sla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb,
ldafb, ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm,
err_bnds_comp, res, ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise,
info )

call dla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb,
ldafb, ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm,
err_bnds_comp, res, ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise,
info )

call cla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb,
ldafb, ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm,
err_bnds_comp, res, ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise,
info )

call zla_gbrfsx_extended( prec_type, trans_type, n, kl, ku, nrhs, ab, ldab, afb,
ldafb, ipiv, colequ, c, b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm,
err_bnds_comp, res, ayb, dy, y_tail, rcond, ithresh, rthresh, dz_ub, ignore_cwise,
info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?la_gbrfsx_extended subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The ?gbrfsx routine calls ?la_gbrfsx_extended to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

Use ?la_gbrfsx_extended to set only the second fields of `err_bnds_norm` and `err_bnds_comp`.

Input Parameters

<code>prec_type</code>	INTEGER.
	Specifies the intermediate precision to be used in refinement. The value is defined by ilaprec(p), where p is a CHARACTER and:
If <code>p = 'S'</code> :	Single.
If <code>p = 'D'</code> :	Double.
If <code>p = 'I'</code> :	Indigenous.
If <code>p = 'X', 'E'</code> :	Extra.

<i>trans_type</i>	INTEGER. Specifies the transposition operation on A . The value is defined by <code>ilatrans(t)</code> , where t is a CHARACTER and: If $t = 'N'$: No transpose. If $t = 'T'$: Transpose. If $t = 'C'$: Conjugate Transpose.
<i>n</i>	INTEGER. The number of linear equations; the order of the matrix A ; $n \geq 0$.
<i>k1</i>	INTEGER. The number of subdiagonals within the band of A ; $k1 \geq 0$.
<i>ku</i>	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the matrix B .
<i>ab, afb, b, y</i>	REAL for <code>sla_gbrfsx_extended</code> DOUBLE PRECISION for <code>dla_gbrfsx_extended</code> COMPLEX for <code>cla_gbrfsx_extended</code> DOUBLE COMPLEX for <code>zla_gbrfsx_extended</code> . Arrays: $ab(1dab, *)$, $afb(1dafb, *)$, $b(1db, *)$, $y(1dy, *)$. The array <i>ab</i> contains the original n -by- n matrix A . The second dimension of <i>ab</i> must be at least $\max(1, n)$. The array <i>afb</i> contains the factors L and U from the factorization $A = P^* L^* U$ as computed by <code>?gbtrf</code> . The second dimension of <i>afb</i> must be at least $\max(1, n)$. The array <i>b</i> contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of <i>b</i> must be at least $\max(1, nrhs)$. The array <i>y</i> on entry contains the solution matrix X as computed by <code>?gbtrs</code> . The second dimension of <i>y</i> must be at least $\max(1, nrhs)$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> ; $1dab \geq \max(1, n)$.
<i>ldafb</i>	INTEGER. The leading dimension of the array <i>afb</i> ; $1dafb \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION at least $\max(1, n)$. Contains the pivot indices from the factorization $A = P^* L^* U$ as computed by <code>?gbtrf</code> ; row i of the matrix was interchanged with row $ipiv(i)$.
<i>colequ</i>	LOGICAL. If <i>colequ</i> = .TRUE., column equilibration was done to A before calling this routine. This is needed to compute the solution and error bounds correctly.
<i>c</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors.

c contains the column scale factors for *A*. If *colequ* = .FALSE., *c* is not accessed.

If *c* is input, each element of *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb INTEGER. The leading dimension of the array *b*; *ldb* $\geq \max(1, n)$.

ldy INTEGER. The leading dimension of the array *y*; *ldy* $\geq \max(1, n)$.

n_norms INTEGER. Determines which error bounds to return. See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.

If *n_norms* ≥ 1 , returns normwise error bounds.

If *n_norms* ≥ 2 , returns componentwise error bounds.

err_bnds_norm REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (*nrhs, n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the *i*-th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in *err_bnds_norm(i,:)* corresponds to the *i*-th right-hand side.

The second index in *err_bnds_norm(:,err)* contains the following three fields:

err=1 "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold *sqrt(n) * slamch(ε)* for single precision flavors and *sqrt(n) * dlamch(ε)* for double precision flavors.

err=2 "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold *sqrt(n) * slamch(ε)* for single precision flavors and *sqrt(n) * dlamch(ε)* for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\varepsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z.

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

`err_bnds_comp`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION ($\text{nrhs}, n_{\text{err_bnds}}$). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{\text{true}_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params}(3) = 0.0$), then `err_bnds_comp` is not accessed. If $n_{\text{err_bnds}} < 3$, then at most the first $(:, n_{\text{err_bnds}})$ entries are returned.

The first index in `err_bnds_comp(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:, err)` contains the following three fields:

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\varepsilon)$ for double precision flavors.

`err=2`

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\varepsilon)$ for single precision flavors and

$\sqrt{n} * \text{diamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{diamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s^* (a^* \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a^* \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

`res, dy, y_tail`

REAL for `sla_gbrfsx_extended`
 DOUBLE PRECISION for `dla_gbrfsx_extended`
 COMPLEX for `cla_gbrfsx_extended`
 DOUBLE COMPLEX for `zla_gbrfsx_extended`.
Workspace arrays of DIMENSION n .
`res` holds the intermediate residual.
`dy` holds the intermediate solution.
`y_tail` holds the trailing bits of the intermediate solution.

`ayb`

REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
Workspace array, DIMENSION n .

`rcond`

REAL for single precision flavors
 DOUBLE PRECISION for double precision flavors.
 Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If `rcond` is less than the machine precision, in particular, if `rcond = 0`, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

`ithresh`

INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

<i>rthresh</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies $\text{norm}(\text{dx}_{\{i+1\}}) < \text{rthresh} * \text{norm}(\text{dx}_i)$ where $\text{norm}(z)$ is the infinity norm of Z . <i>rthresh</i> satisfies $0 < \text{rthresh} \leq 1$.
<i>dz_ub</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Determines when to start considering componentwise convergence. Componentwise <i>dz_ub</i> convergence is only considered after each component of the solution y is stable, that is, the relative change in each component is less than <i>dz_ub</i> . The default value is 0.25, requiring the first bit to be stable.
<i>ignore_cwise</i>	LOGICAL If .TRUE., the function ignores componentwise convergence. Default value is .FALSE..

Output Parameters

<i>y</i>	REAL for <code>sla_gbrfsx_extended</code> DOUBLE PRECISION for <code>dla_gbrfsx_extended</code> COMPLEX for <code>cla_gbrfsx_extended</code> DOUBLE COMPLEX for <code>zla_gbrfsx_extended</code> . The improved solution matrix Y .
<i>berr_out</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least <code>max(1, nrhs)</code> . Contains the componentwise relative backward error for right-hand-side j from the formula $\max(i) (\text{abs}(\text{res}(i)) / (\text{abs}(\text{op}(A)) * \text{abs}(y) + \text{abs}(B)) (i))$ where $\text{abs}(z)$ is the componentwise absolute value of the matrix or vector Z . This is computed by <code>?la_lin_berr</code> .
<i>err_bnds_norm</i> , <i>err_bnds_comp</i>	Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array (<code>1:nrhs, 2</code>). The other elements are kept unchanged.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If $info = -i$, the i -th parameter had an illegal value.

See Also

[?gbrfsx](#)
[?gbtrf](#)
[?gbtrs](#)
[?lamch](#)
[ilaprec](#)
[ilatrans](#)
[?la_lin_berr](#)

?la_gbrpvgrw

Computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$ for a general band matrix.

Syntax

FORTRAN 77:

```
call sla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
call dla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
call cla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
call zla_gbrpvgrw( n, kl, ku, ncols, ab, ldab, afb, ldafb )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?la_gbrpvgrw routine computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element* norm is used. If this is much less than 1, the stability of the LU factorization of the equilibrated matrix A could be poor. This also means that the solution X , estimated condition numbers, and error bounds could be unreliable.

Input Parameters

n	INTEGER. The number of linear equations, the order of the matrix A ; $n \geq 0$.
kl	INTEGER. The number of subdiagonals within the band of A ; $kl \geq 0$.
ku	INTEGER. The number of superdiagonals within the band of A ; $ku \geq 0$.
$ncols$	INTEGER. The number of columns of the matrix A ; $ncols \geq 0$.
ab, afb	REAL for <code>sla_gbrpvgrw</code> DOUBLE PRECISION for <code>dla_gbrpvgrw</code> COMPLEX for <code>cla_gbrpvgrw</code> DOUBLE COMPLEX for <code>zla_gbrpvgrw</code> . Arrays: $ab(ldab, *)$, $afb(ldafb, *)$.

ab contains the original band matrix *A* (see [Matrix Storage Schemes](#)) stored in rows from 1 to $k_l + k_u + 1$. The j -th column of *A* is stored in the j -th column of the array *ab* as follows:

$ab(k_l+1+i-j, j) = A(i, j)$

for

$\max(1, j-k_u) \leq i \leq \min(n, j+k_l)$

afb contains details of the LU factorization of the band matrix *A*, as returned by [?gbtrf](#). *U* is stored as an upper triangular band matrix with k_l+k_u superdiagonals in rows 1 to k_l+k_u+1 , and the multipliers used during the factorization are stored in rows k_l+k_u+2 to $2*k_l+k_u+1$.

ldab

INTEGER. The leading dimension of *ab*; $ldab \geq k_l+k_u+1$.

ldafb

INTEGER. The leading dimension of *afb*; $ldafb \geq 2*k_l+k_u+1$.

See Also

[?gbtrf](#)

?la_geamv

Computes a matrix-vector product using a general matrix to calculate error bounds.

Syntax

FORTRAN 77:

```
call sla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call dla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call cla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
call zla_geamv(trans, m, n, alpha, a, lda, x, incx, beta, y, incy)
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The `?la_geamv` routines perform a matrix-vector operation defined as

$y := \alpha * \text{abs}(A) * (x) + \beta * \text{abs}(y),$

or

$y := \alpha * \text{abs}(A^T) * \text{abs}(x) + \beta * \text{abs}(y),$

where:

α and β are scalars,

x and y are vectors,

A is an m -by- n matrix.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by $(n + 1)$ times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

<i>trans</i>	CHARACTER*1. Specifies the operation:
	if <i>trans</i> = BLAS_NO_TRANS , then <i>y</i> := <i>alpha</i> *abs(<i>A</i>)*abs(<i>x</i>) + <i>beta</i> *abs(<i>y</i>)
	if <i>trans</i> = BLAS_TRANS, then <i>y</i> := <i>alpha</i> *abs(<i>A</i> ^T)*abs(<i>x</i>) + <i>beta</i> *abs(<i>y</i>)
	if <i>trans</i> = 'BLAS_CONJ_TRANS, then <i>y</i> := <i>alpha</i> *abs(<i>A</i> ^T)*abs(<i>x</i>) + <i>beta</i> *abs(<i>y</i>).
<i>m</i>	INTEGER. Specifies the number of rows of the matrix <i>A</i> . The value of <i>m</i> must be at least zero.
<i>n</i>	INTEGER. Specifies the number of columns of the matrix <i>A</i> . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for <i>sla_geamv</i> and for <i>cla_geamv</i> DOUBLE PRECISION for <i>dla_geamv</i> and <i>zla_geamv</i> Specifies the scalar <i>alpha</i> .
<i>a</i>	REAL for <i>sla_geamv</i> DOUBLE PRECISION for <i>dla_geamv</i> COMPLEX for <i>cla_geamv</i> DOUBLE COMPLEX for <i>zla_geamv</i> Array, DIMENSION (<i>lda</i> , *). Before entry, the leading <i>m</i> -by- <i>n</i> part of the array <i>a</i> must contain the matrix of coefficients. The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, m)$.
<i>x</i>	REAL for <i>sla_geamv</i> DOUBLE PRECISION for <i>dla_geamv</i> COMPLEX for <i>cla_geamv</i> DOUBLE COMPLEX for <i>zla_geamv</i> Array, DIMENSION at least $(1+(n-1)*\text{abs}(incx))$ when <i>trans</i> = 'N' or 'n' and at least $(1+(m - 1)*\text{abs}(incx))$ otherwise. Before entry, the incremented array <i>x</i> must contain the vector <i>X</i> .
<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must be non-zero.
<i>beta</i>	REAL for <i>sla_geamv</i> and for <i>cla_geamv</i>

DOUBLE PRECISION for `dla_geamv` and `zla_geamv`

Specifies the scalar *beta*. When *beta* is zero, you do not need to set *y* on input.

y

REAL for `sla_geamv` and for `cla_geamv`

DOUBLE PRECISION for `dla_geamv` and `zla_geamv`

Array, DIMENSION at least $(1 + (m - 1) * \text{abs}(\text{incy}))$ when *trans* = 'N' or 'n' and at least $(1 + (n - 1) * \text{abs}(\text{incy}))$ otherwise. Before entry with non-zero *beta*, the incremented array *y* must contain the vector *Y*.

incy

INTEGER. Specifies the increment for the elements of *y*.

The value of *incy* must be non-zero.

Output Parameters

y

Updated vector *Y*.

?la_gercond

Estimates the Skeel condition number for a general matrix.

Syntax

FORTRAN 77:

```
call sla_gercond( trans, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
call dla_gercond( trans, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The function estimates the Skeel condition number of

$\text{op}(A) * \text{op2}(C)$

where

the *cmode* parameter determines op2 as follows:

cmode Value	op2(C)
1	C
0	I
-1	$\text{inv}(C)$

The Skeel condition number

$\text{cond}(A) = \text{norminf}(|\text{inv}(A)| |A|)$

is computed by computing scaling factors *R* such that

$\text{diag}(R) * A * \text{op2}(C)$

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

<i>trans</i>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Specifies the form of the system of equations: If <i>trans</i> = 'N', the system has the form $A*X = B$ (No transpose). If <i>trans</i> = 'T', the system has the form $A^T*X = B$ (Transpose). If <i>trans</i> = 'C', the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose).
<i>n</i>	INTEGER. The number of linear equations, that is, the order of the matrix <i>A</i> ; $n \geq 0$.
<i>a</i> , <i>af</i> , <i>c</i> , <i>work</i>	REAL for <code>sla_gercond</code> DOUBLE PRECISION for <code>dla_gercond</code> Arrays: <i>a</i> (<i>lda</i> ,*) contains the original general <i>n</i> -by- <i>n</i> matrix <i>A</i> . <i>af</i> (<i>ldaf</i> ,*) contains factors <i>L</i> and <i>U</i> from the factorization of the general matrix $A = P*L*U$, as returned by <code>?getrf</code> . <i>c</i> , DIMENSION <i>n</i> . The vector <i>C</i> in the formula $\text{op}(A) * \text{op}2(C)$. <i>work</i> is a workspace array of DIMENSION ($3*n$). The second dimension of <i>a</i> and <i>af</i> must be at least $\max(1, n)$. <i>lda</i> INTEGER. The leading dimension of the array <i>a</i> . $lda \geq \max(1, n)$. <i>ldaf</i> INTEGER. The leading dimension of <i>af</i> . $ldaf \geq \max(1, n)$. <i>ipiv</i> INTEGER. Array with DIMENSION <i>n</i> . The pivot indices from the factorization $A = P*L*U$ as computed by <code>?getrf</code> . Row <i>i</i> of the matrix was interchanged with row <i>ipiv(i)</i> . <i>cmode</i> INTEGER. Determines $\text{op}2(C)$ in the formula $\text{op}(A) * \text{op}2(C)$ as follows: If <i>cmode</i> = 1, $\text{op}2(C) = C$. If <i>cmode</i> = 0, $\text{op}2(C) = I$. If <i>cmode</i> = -1, $\text{op}2(C) = \text{inv}(C)$. <i>iwork</i> INTEGER. Workspace array with DIMENSION <i>n</i> .

Output Parameters

<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If <i>i</i> > 0, the <i>i</i> -th parameter is invalid.
-------------	---

See Also

`?getrf`

?la_gercond_c

Computes the infinity norm condition number of $\text{op}(A)^*\text{inv}(\text{diag}(c))$ for general matrices.

Syntax

FORTRAN 77:

```
call cla_gercond_c( trans, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
call zla_gercond_c( trans, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function computes the infinity norm condition number of

$\text{op}(A)^* \text{inv}(\text{diag}(c))$

where the c is a REAL vector for `cla_gercond_c` and a DOUBLE PRECISION vector for `zla_gercond_c`.

Input Parameters

<code>trans</code>	CHARACTER*1. Must be 'N' or 'T' or 'C'. Specifies the form of the system of equations: If <code>trans</code> = 'N', the system has the form $A*X = B$ (No transpose) If <code>trans</code> = 'T', the system has the form $A^T*X = B$ (Transpose) If <code>trans</code> = 'C', the system has the form $A^H*X = B$ (Conjugate Transpose = Transpose)
<code>n</code>	INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.
<code>a, af, work</code>	COMPLEX for <code>cla_gercond_c</code> DOUBLE COMPLEX for <code>zla_gercond_c</code> Arrays: $a(lda,*)$ contains the original general n -by- n matrix A . $af(ldaf,*)$ contains the factors L and U from the factorization $A = P^* L^* U$ as returned by <code>?getrf</code> . <code>work</code> is a workspace array of DIMENSION $(2*n)$. The second dimension of <code>a</code> and <code>af</code> must be at least $\max(1, n)$.
<code>lda</code>	INTEGER. The leading dimension of the array <code>a</code> . $lda \geq \max(1, n)$.
<code>ldaf</code>	INTEGER. The leading dimension of <code>af</code> . $ldaf \geq \max(1, n)$.
<code>ipiv</code>	INTEGER.

Array with DIMENSION n . The pivot indices from the factorization $A = P^* L^* U$ as computed by `?getrf`. Row i of the matrix was interchanged with row $ipiv(i)$.

$c, rwork$

REAL for `cla_gercond_c`
 DOUBLE PRECISION for `zla_gercond_c`
 Array c with DIMENSION n . The vector c in the formula
 $\text{op}(A) * \text{inv}(\text{diag}(c))$.
 Array $rwork$ with DIMENSION n is a workspace.

$capply$

LOGICAL. If $capply = .TRUE.$, then the function uses the vector c from the formula
 $\text{op}(A) * \text{inv}(\text{diag}(c))$.

Output Parameters

$info$

INTEGER.
 If $info = 0$, the execution is successful.
 If $i > 0$, the i -th parameter is invalid.

See Also

`?getrf`

`?la_gercond_x`

Computes the infinity norm condition number of $\text{op}(A)*\text{diag}(x)$ for general matrices.

Syntax

FORTRAN 77:

```
call cla_gercond_x( trans, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
call zla_gercond_x( trans, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The function computes the infinity norm condition number of

$\text{op}(A) * \text{diag}(x)$

where the x is a COMPLEX vector for `cla_gercond_x` and a DOUBLE COMPLEX vector for `zla_gercond_x`.

Input Parameters

$trans$

CHARACTER*1. Must be 'N' or 'T' or 'C'.
 Specifies the form of the system of equations:
 If $trans = 'N'$, the system has the form $A*X = B$ (No transpose)

If $\text{trans} = \text{'T'}$, the system has the form $A^T X = B$ (Transpose)

If $\text{trans} = \text{'C'}$, the system has the form $A^H X = B$ (Conjugate Transpose = Transpose)

n INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.

a, af, x, work COMPLEX for `cla_gercond_x`

DOUBLE COMPLEX for `zla_gercond_x`

Arrays:

a(*lda,**) contains the original general n -by- n matrix A .

af(*ldaf,**) contains the factors L and U from the factorization $A = P^* L^* U$ as returned by `?getrf`.

x, DIMENSION n . The vector *x* in the formula $\text{op}(A) * \text{diag}(x)$.

work is a workspace array of DIMENSION ($2*n$).

The second dimension of *a* and *af* must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of the array *a*. $\text{lda} \geq \max(1, n)$.

ldaf INTEGER. The leading dimension of *af*. $\text{ldaf} \geq \max(1, n)$.

ipiv INTEGER.

Array with DIMENSION n . The pivot indices from the factorization $A = P^* L^* U$ as computed by `?getrf`. Row i of the matrix was interchanged with row *ipiv*(i).

rwork REAL for `cla_gercond_x`

DOUBLE PRECISION for `zla_gercond_x`

Array *rwork* with DIMENSION n is a workspace.

Output Parameters

info INTEGER.

If *info* = 0, the execution is successful.

If $i > 0$, the i -th parameter is invalid.

See Also

`?getrf`

`?la_gerfsx_extended`

Improves the computed solution to a system of linear equations for general matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

FORTRAN 77:

```

call sla_gerfsx_extended( prec_type, trans_type, n, nrhs, a, lda, af, ldaf, ipiv,
colequ, c, b, ldb, y, ldy, berr_out, n_norms, errs_n, errs_c, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )

call dla_gerfsx_extended( prec_type, trans_type, n, nrhs, a, lda, af, ldaf, ipiv,
colequ, c, b, ldb, y, ldy, berr_out, n_norms, errs_n, errs_c, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )

call cla_gerfsx_extended( prec_type, trans_type, n, nrhs, a, lda, af, ldaf, ipiv,
colequ, c, b, ldb, y, ldy, berr_out, n_norms, errs_n, errs_c, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )

call zla_gerfsx_extended( prec_type, trans_type, n, nrhs, a, lda, af, ldaf, ipiv,
colequ, c, b, ldb, y, ldy, berr_out, n_norms, errs_n, errs_c, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?la_gerfsx_extended subroutine improves the computed solution to a system of linear equations for general matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The ?gerfsx routine calls ?la_gerfsx_extended to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for `errs_n` and `errs_c` for details of the error bounds.

Use ?la_gerfsx_extended to set only the second fields of `errs_n` and `errs_c`.

Input Parameters

<code>prec_type</code>	INTEGER. Specifies the intermediate precision to be used in refinement. The value is defined by <code>ilaprec(p)</code> , where <code>p</code> is a CHARACTER and: If <code>p = 'S'</code> : Single. If <code>p = 'D'</code> : Double. If <code>p = 'I'</code> : Indigenous. If <code>p = 'X', 'E'</code> : Extra.
<code>trans_type</code>	INTEGER. Specifies the transposition operation on <code>A</code> . The value is defined by <code>ilatrans(t)</code> , where <code>t</code> is a CHARACTER and: If <code>t = 'N'</code> : No transpose. If <code>t = 'T'</code> : Transpose. If <code>t = 'C'</code> : Conjugate Transpose.
<code>n</code>	INTEGER. The number of linear equations; the order of the matrix <code>A</code> ; $n \geq 0$.
<code>nrhs</code>	INTEGER. The number of right-hand sides; the number of columns of the matrix <code>B</code> .

a, af, b, y

REAL for `sla_gerfsx_extended`
 DOUBLE PRECISION for `dla_gerfsx_extended`
 COMPLEX for `cla_gerfsx_extended`
 DOUBLE COMPLEX for `zla_gerfsx_extended`.
Arrays: $a(1da,*)$, $af(1daf,*)$, $b(1db,*)$, $y(1dy,*)$.

The array a contains the original matrix n -by- n matrix A . The second dimension of a must be at least $\max(1, n)$.

The array af contains the factors L and U from the factorization $A = P^*L^*U$ as computed by `?getrf`. The second dimension of af must be at least $\max(1, n)$.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations. The second dimension of b must be at least $\max(1, nrhs)$.

The array y on entry contains the solution matrix X as computed by `?getrs`. The second dimension of y must be at least $\max(1, nrhs)$.

lda

INTEGER. The leading dimension of the array a ; $lda \geq \max(1, n)$.

ldaf

INTEGER. The leading dimension of the array af ; $ldaf \geq \max(1, n)$.

ipiv

INTEGER.

Array, DIMENSION at least $\max(1, n)$. Contains the pivot indices from the factorization $A = P^*L^*U$ as computed by `?getrf`; row i of the matrix was interchanged with row $ipiv(i)$.

colequ

LOGICAL. If $colequ = .TRUE.$, column equilibration was done to A before calling this routine. This is needed to compute the solution and error bounds correctly.

c

REAL for single precision flavors (`sla_gerfsx_extended`,
`cla_gerfsx_extended`)
 DOUBLE PRECISION for double precision flavors (`dla_gerfsx_extended`,
`zla_gerfsx_extended`).
 c contains the column scale factors for A . If $colequ = .FALSE.$, c is not used.

If c is input, each element of c should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array b ; $ldb \geq \max(1, n)$.

ldy

INTEGER. The leading dimension of the array y ; $ldy \geq \max(1, n)$.

n_norms

INTEGER. Determines which error bounds to return. See `errs_n` and `errs_c` descriptions in *Output Arguments* section below.

If $n_norms \geq 1$, returns normwise error bounds.

If $n_norms \geq 2$, returns componentwise error bounds.

`errs_n`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION ($nrhs, n_err_bnds$). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the i -th solution vector

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `errs_n(i, :)` corresponds to the i -th right-hand side.

The second index in `errs_n(:, err)` contains the following three fields:

`err=1` "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

`err=3` Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix Z .

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

`errs_c`

REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Array, DIMENSION (*nrhs, n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the *i*-th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side *i*, on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (*params(3) = 0.0*), then *errs_c* is not accessed. If *n_err_bnds < 3*, then at most the first *(:,n_err_bnds)* entries are returned.

The first index in *errs_c(i,:)* corresponds to the *i*-th right-hand side.

The second index in *errs_c(:,err)* contains the following three fields:

<i>err=1</i>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors.
<i>err=2</i>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.
<i>err=3</i>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * slamch(\epsilon)$ for single precision flavors and $\sqrt{n} * dlamch(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix <i>Z</i> . Let $z = s * (a * \text{diag}(x))$, where <i>x</i> is the solution for the current right-hand side and <i>s</i> scales each row of <i>a * diag(x)</i> by a power of the radix so all absolute row sums of <i>z</i> are approximately 1. Use this subroutine to set only the second field above.

<i>res, dy, y_tail</i>	REAL for <code>sla_gerfsx_extended</code> DOUBLE PRECISION for <code>dla_gerfsx_extended</code> COMPLEX for <code>cla_gerfsx_extended</code> DOUBLE COMPLEX for <code>zla_gerfsx_extended</code> . Workspace arrays of DIMENSION <i>n</i> . <i>res</i> holds the intermediate residual. <i>dy</i> holds the intermediate solution. <i>y_tail</i> holds the trailing bits of the intermediate solution.
<i>ayb</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Workspace array, DIMENSION <i>n</i> .
<i>rcond</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision, in particular, if <i>rcond</i> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.
<i>ithresh</i>	INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in <i>errs_n</i> and <i>errs_c</i> may no longer be trustworthy.
<i>rthresh</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies $\text{norm}(\text{dx}_{\{i+1\}}) < \text{rthresh} * \text{norm}(\text{dx}_i)$ where $\text{norm}(z)$ is the infinity norm of <i>Z</i> . <i>rthresh</i> satisfies $0 < \text{rthresh} \leq 1$. The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.
<i>dz_ub</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Determines when to start considering componentwise convergence. Componentwise <i>dz_ub</i> convergence is only considered after each component of the solution <i>y</i> is stable, that is, the relative change in each component is less than <i>dz_ub</i> . The default value is 0.25, requiring the first bit to be stable.

ignore_cwise LOGICAL
If .TRUE., the function ignores componentwise convergence. Default value is .FALSE.

Output Parameters

<i>y</i>	REAL for <code>sla_gerfsx_extended</code> DOUBLE PRECISION for <code>dla_gerfsx_extended</code> COMPLEX for <code>cla_gerfsx_extended</code> DOUBLE COMPLEX for <code>zla_gerfsx_extended</code> .
<i>berr_out</i>	The improved solution matrix <i>Y</i> .
<i>errs_n, errs_c</i>	REAL for single precision flavors DOUBLE PRECISION for double precision flavors. Array, DIMENSION at least <code>max(1, nrhs)</code> . Contains the componentwise relative backward error for right-hand-side <i>j</i> from the formula $\max(i) \left(\frac{\ res(i)\ }{\ op(A)\ \ y\ + \ B\ } \right)$ where <code>abs(z)</code> is the componentwise absolute value of the matrix or vector <i>Z</i> . This is computed by <code>?la_lin_berr</code> .
<i>info</i>	Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array <code>(1:nrhs, 2)</code> . The other elements are kept unchanged.
	INTEGER. If <i>info</i> = 0, the execution is successful. The solution to every right-hand side is guaranteed. If <i>info</i> = <i>-i</i> , the <i>i</i> -th parameter had an illegal value.

See Also

?gerfsx
?getrf
sgetrs dgetrs cgetrs zgetrs Reference
?lamch
ilaprec
ilatrans
?la lin berr

?la heamv

Computes a matrix-vector product using a Hermitian indefinite matrix to calculate error bounds.

Syntax

FORTRAN 77:

```
call cla_heamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)  
call zla_heamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?la_heamv routines perform a matrix-vector operation defined as

$$y := \alpha * \text{abs}(A) * \text{abs}(x) + \beta * \text{abs}(y),$$

where:

α and β are scalars,

x and y are vectors,

A is an n -by- n Hermitian matrix.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by $(n + 1)$ times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the upper or lower triangular part of the array A is to be referenced: If <i>uplo</i> = 'BLAS_UPPER', only the upper triangular part of A is to be referenced, If <i>uplo</i> = 'BLAS_LOWER', only the lower triangular part of A is to be referenced.
<i>n</i>	INTEGER. Specifies the number of rows and columns of the matrix A . The value of <i>n</i> must be at least zero.
<i>alpha</i>	REAL for <code>cla_heamv</code> DOUBLE PRECISION for <code>zla_heamv</code> Specifies the scalar α .
<i>a</i>	COMPLEX for <code>cla_heamv</code> DOUBLE COMPLEX for <code>zla_heamv</code> Array, DIMENSION (<i>lda</i> , *). Before entry, the leading m -by- n part of the array <i>a</i> must contain the matrix of coefficients. The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. Specifies the leading dimension of <i>a</i> as declared in the calling (sub)program. The value of <i>lda</i> must be at least $\max(1, n)$.
<i>x</i>	COMPLEX for <code>cla_heamv</code> DOUBLE COMPLEX for <code>zla_heamv</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(\text{incx}))$. Before entry, the incremented array <i>x</i> must contain the vector <i>X</i> .

<i>incx</i>	INTEGER. Specifies the increment for the elements of <i>x</i> . The value of <i>incx</i> must be non-zero.
<i>beta</i>	REAL for <code>cla_heamv</code> DOUBLE PRECISION for <code>zla_heamv</code> Specifies the scalar <i>beta</i> . When <i>beta</i> is zero, you do not need to set <i>y</i> on input.
<i>y</i>	REAL for <code>cla_heamv</code> DOUBLE PRECISION for <code>zla_heamv</code> Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incy))$ otherwise. Before entry with non-zero <i>beta</i> , the incremented array <i>y</i> must contain the vector <i>Y</i> .
<i>incy</i>	INTEGER. Specifies the increment for the elements of <i>y</i> . The value of <i>incy</i> must be non-zero.

Output Parameters

<i>y</i>	Updated vector <i>Y</i> .
----------	---------------------------

?la_hercond_c

Computes the infinity norm condition number of $\text{op}(A)^* \text{inv}(\text{diag}(c))$ for Hermitian indefinite matrices.

Syntax

FORTRAN 77:

```
call cla_hercond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
call zla_hercond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The function computes the infinity norm condition number of

$\text{op}(A)^* \text{inv}(\text{diag}(c))$

where the *c* is a REAL vector for `cla_hercond_c` and a DOUBLE PRECISION vector for `zla_hercond_c`.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of <i>A</i> to store: If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored, If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
-------------	--

<i>n</i>	INTEGER. The number of linear equations, that is, the order of the matrix <i>A</i> ; $n \geq 0$.
<i>a</i>	COMPLEX for <code>cla_hercond_c</code> DOUBLE COMPLEX for <code>zla_hercond_c</code> Array, DIMENSION (<i>lda</i> , *). On entry, the n -by- n matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . $\text{lda} \geq \max(1, n)$.
<i>af</i>	COMPLEX for <code>cla_hercond_c</code> DOUBLE COMPLEX for <code>zla_hercond_c</code> Array, DIMENSION (<i>ldaf</i> , *). The block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> as computed by <code>?hetrf</code> . The second dimension of <i>af</i> must be at least $\max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of the array <i>af</i> . $\text{ldaf} \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array with DIMENSION <i>n</i> . Details of the interchanges and the block structure of <i>D</i> as determined by <code>?hetrf</code> .
<i>c</i>	REAL for <code>cla_hercond_c</code> DOUBLE PRECISION for <code>zla_hercond_c</code> Array <i>c</i> with DIMENSION <i>n</i> . The vector <i>c</i> in the formula $\text{op}(A) * \text{inv}(\text{diag}(c))$.
<i>capply</i>	LOGICAL. If .TRUE., then the function uses the vector <i>c</i> from the formula $\text{op}(A) * \text{inv}(\text{diag}(c))$.
<i>work</i>	COMPLEX for <code>cla_hercond_c</code> DOUBLE COMPLEX for <code>zla_hercond_c</code> Array DIMENSION $2*n$. Workspace.
<i>rwork</i>	REAL for <code>cla_hercond_c</code> DOUBLE PRECISION for <code>zla_hercond_c</code> Array DIMENSION <i>n</i> . Workspace.

Output Parameters

<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. If $i > 0$, the <i>i</i> -th parameter is invalid.
-------------	---

See Also

`?hetrf`

?la_hercond_x

Computes the infinity norm condition number of
 $\text{op}(A) * \text{diag}(x)$ for Hermitian indefinite matrices.

Syntax

FORTRAN 77:

```
call cla_hercond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
call zla_hercond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function computes the infinity norm condition number of

$\text{op}(A) * \text{diag}(x)$

where the x is a COMPLEX vector for `cla_hercond_x` and a DOUBLE COMPLEX vector for `zla_hercond_x`.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If <code>uplo</code> = 'U', the upper triangle of A is stored, If <code>uplo</code> = 'L', the lower triangle of A is stored.
<code>n</code>	INTEGER. The number of linear equations, that is, the order of the matrix A; $n \geq 0$.
<code>a</code>	COMPLEX for <code>cla_hercond_c</code> DOUBLE COMPLEX for <code>zla_hercond_c</code> Array, DIMENSION (<code>lda, *</code>). On entry, the n -by- n matrix A. The second dimension of <code>a</code> must be at least $\max(1, n)$.
<code>lda</code>	INTEGER. The leading dimension of the array <code>a</code> . $\text{lda} \geq \max(1, n)$.
<code>af</code>	COMPLEX for <code>cla_hercond_c</code> DOUBLE COMPLEX for <code>zla_hercond_c</code> Array, DIMENSION (<code>ldaf, *</code>). The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by <code>?hetrf</code> . The second dimension of <code>af</code> must be at least $\max(1, n)$.
<code>ldaf</code>	INTEGER. The leading dimension of the array <code>af</code> . $\text{ldaf} \geq \max(1, n)$.
<code>ipiv</code>	INTEGER. Array with DIMENSION n . Details of the interchanges and the block structure of D as determined by <code>?hetrf</code> .

<i>x</i>	COMPLEX for <code>cla_hercond_c</code> DOUBLE COMPLEX for <code>zla_hercond_c</code>
	Array <i>x</i> with DIMENSION <i>n</i> . The vector <i>x</i> in the formula $\text{op}(A) * \text{inv}(\text{diag}(x))$.
<i>work</i>	COMPLEX for <code>cla_hercond_c</code> DOUBLE COMPLEX for <code>zla_hercond_c</code>
	Array DIMENSION $2*n$. Workspace.
<i>rwork</i>	REAL for <code>cla_hercond_c</code> DOUBLE PRECISION for <code>zla_hercond_c</code>
	Array DIMENSION <i>n</i> . Workspace.

Output Parameters

info INTEGER.
If *info* = 0, the execution is successful.
If *i* > 0, the *i*-th parameter is invalid.

See Also

[?hetrf](#)

?la_herfsx_extended

Improves the computed solution to a system of linear equations for Hermitian indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

FORTRAN 77:

```
call cla_herfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
call zla_herfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The `?la_herfsx_extended` subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The `?herfsx` routine calls `?la_herfsx_extended` to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

Use `?la_herfsx_extended` to set only the second fields of `err_bnds_norm` and `err_bnds_comp`.

Input Parameters

`prec_type`

INTEGER.

Specifies the intermediate precision to be used in refinement. The value is defined by `ilaprec(p)`, where `p` is a CHARACTER and:

If `p = 'S'`: Single.

If `p = 'D'`: Double.

If `p = 'I'`: Indigenous.

If `p = 'X', 'E'`: Extra.

`uplo`

CHARACTER*1. Must be '`U`' or '`L`'.

Specifies the triangle of `A` to store:

If `uplo = 'U'`, the upper triangle of `A` is stored,

If `uplo = 'L'`, the lower triangle of `A` is stored.

`n`

INTEGER. The number of linear equations; the order of the matrix `A`; $n \geq 0$.

`nrhs`

INTEGER. The number of right-hand sides; the number of columns of the matrix `B`.

`a, af, b, y`

COMPLEX for `cla_herfsx_extended`

DOUBLE COMPLEX for `zla_herfsx_extended`.

Arrays: `a(1da,*), af(1daf,*), b(1db,*), y(1dy,*)`.

The array `a` contains the original n -by- n matrix `A`. The second dimension of `a` must be at least `max(1,n)`.

The array `af` contains the block diagonal matrix `D` and the multipliers used to obtain the factor `U` or `L` as computed by `?hetrf`. The second dimension of `af` must be at least `max(1,n)`.

The array `b` contains the right-hand-side of the matrix `B`. The second dimension of `b` must be at least `max(1,nrhs)`.

The array `y` on entry contains the solution matrix `X` as computed by `?hetrs`. The second dimension of `y` must be at least `max(1,nrhs)`.

`lda`

INTEGER. The leading dimension of the array `a`; `lda` $\geq \max(1,n)$.

`ldaf`

INTEGER. The leading dimension of the array `af`; `ldaf` $\geq \max(1,n)$.

`ipiv`

INTEGER.

Array, DIMENSION `n`. Details of the interchanges and the block structure of `D` as determined by `?hetrf`.

colequ LOGICAL. If *colequ* = .TRUE., column equilibration was done to *A* before calling this routine. This is needed to compute the solution and error bounds correctly.

c REAL for *cla_herfsx_extended*
 DOUBLE PRECISION for *zla_herfsx_extended*.
c contains the column scale factors for *A*. If *colequ* = .FALSE., *c* is not used.
 If *c* is input, each element of *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb INTEGER. The leading dimension of the array *b*; *ldb* $\geq \max(1, n)$.

ldy INTEGER. The leading dimension of the array *y*; *ldy* $\geq \max(1, n)$.

n_norms INTEGER. Determines which error bounds to return. See *err_bnds_norm* and *err_bnds_comp* descriptions in *Output Arguments* section below.
 If *n_norms* ≥ 1 , returns normwise error bounds.
 If *n_norms* ≥ 2 , returns componentwise error bounds.

err_bnds_norm REAL for *cla_herfsx_extended*
 DOUBLE PRECISION for *zla_herfsx_extended*.
 Array, DIMENSION (*nrhs, n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error.
 Normwise relative error in the *i*-th solution vector is defined as follows:

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.
 The first index in *err_bnds_norm(i, :)* corresponds to the *i*-th right-hand side.
 The second index in *err_bnds_norm(:, err)* contains the following three fields:

<i>err=1</i>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold <i>sqrt(n) * slamch(ε)</i> for <i>cla_herfsx_extended</i> and <i>sqrt(n) * dlamch(ε)</i> for <i>zla_herfsx_extended</i> .
--------------	---

`err=2` "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for `cla_herfsx_extended` and $\sqrt{n} * \text{dlamch}(\epsilon)$ for `zla_herfsx_extended`. This error bound should only be trusted if the previous boolean is true.

`err=3` Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for `cla_herfsx_extended` and $\sqrt{n} * \text{dlamch}(\epsilon)$ for `zla_herfsx_extended` to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix Z .

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

`err_bnds_comp`

REAL for `cla_herfsx_extended`
DOUBLE PRECISION for `zla_herfsx_extended`.
Array, DIMENSION ($nrhs, n_{\text{err_bnds}}$). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (`params(3) = 0.0`), then `err_bnds_comp` is not accessed. If $n_{\text{err_bnds}} < 3$, then at most the first $(:, n_{\text{err_bnds}})$ entries are returned.

The first index in `err_bnds_comp(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:, err)` contains the following three fields:

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>cla_herfsx_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>zla_herfsx_extended</code> .
<code>err=2</code>	"Guaranteed" error bbound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>cla_herfsx_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>zla_herfsx_extended</code> . This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>cla_herfsx_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>zla_herfsx_extended</code> to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix Z . Let $z = s^* (a^* \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a^* \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1. Use this subroutine to set only the second field above.
<code>res, dy, y_tail</code>	COMPLEX for <code>cla_herfsx_extended</code> DOUBLE COMPLEX for <code>zla_herfsx_extended</code> . Workspace arrays of DIMENSION n . <code>res</code> holds the intermediate residual. <code>dy</code> holds the intermediate solution. <code>y_tail</code> holds the trailing bits of the intermediate solution.
<code>ayb</code>	REAL for <code>cla_herfsx_extended</code> DOUBLE PRECISION for <code>zla_herfsx_extended</code> . Workspace array, DIMENSION n .
<code>rcond</code>	REAL for <code>cla_herfsx_extended</code> DOUBLE PRECISION for <code>zla_herfsx_extended</code> .

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

ithresh

INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

rthresh

REAL for `cla_herfsx_extended`

DOUBLE PRECISION for `zla_herfsx_extended`.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

$\text{norm}(\text{dx}_{\{i+1\}}) < rthresh * \text{norm}(\text{dx}_i)$

where $\text{norm}(z)$ is the infinity norm of Z .

rthresh satisfies

$0 < rthresh \leq 1$.

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub

REAL for `cla_herfsx_extended`

DOUBLE PRECISION for `zla_herfsx_extended`.

Determines when to start considering componentwise convergence.

Componentwise *dz_ub* convergence is only considered after each component of the solution y is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise

LOGICAL

If .TRUE., the function ignores componentwise convergence. Default value is .FALSE..

Output Parameters

y

COMPLEX for `cla_herfsx_extended`

DOUBLE COMPLEX for `zla_herfsx_extended`.

The improved solution matrix Y .

berr_out

REAL for `cla_herfsx_extended`

DOUBLE PRECISION for `zla_herfsx_extended`.

Array, DIMENSION $nrhs$. *berr_out*(j) contains the componentwise relative backward error for right-hand-side j from the formula

$$\max(i) (\text{abs}(\text{res}(i)) / (\text{abs}(\text{op}(A)) * \text{abs}(y) + \text{abs}(B))(i))$$

where $\text{abs}(z)$ is the componentwise absolute value of the matrix or vector Z . This is computed by `?la_lin_berr`.

`err_bnds_norm`,
`err_bnds_comp`

Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array (`1:nrhs, 2`). The other elements are kept unchanged.

`info`

INTEGER. If $info = 0$, the execution is successful. The solution to every right-hand side is guaranteed.
If $info = -i$, the i -th parameter had an illegal value.

See Also

[?herfsx](#)
[?hetrf](#)
[?hetrs](#)
[?lamch](#)
[ilaprec](#)
[ilatrans](#)
[?la_lin_berr](#)

?la_herpvgrw

Computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$ for a Hermitian indefinite matrix.

Syntax

FORTRAN 77:

```
call cla_herpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
call zla_herpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The `?la_herpvgrw` routine computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element norm* is used. If this is much less than 1, the stability of the *LU* factorization of the equilibrated matrix A could be poor. This also means that the solution X , estimated condition numbers, and error bounds could be unreliable.

Input Parameters

`uplo`

CHARACTER*1. Must be '`U`' or '`L`'.
Specifies the triangle of A to store:
If $uplo = 'U'$, the upper triangle of A is stored,
If $uplo = 'L'$, the lower triangle of A is stored.

`n`

INTEGER. The number of linear equations, the order of the matrix A ; $n \geq 0$.

<i>info</i>	INTEGER. The value of INFO returned from ?hetrf, that is, the pivot in column <i>info</i> is exactly 0.
<i>a, af</i>	COMPLEX for cla_herpgrw DOUBLE COMPLEX for zla_herpgrw. Arrays: <i>a</i> (<i>lda</i> , *), <i>af</i> (<i>ldaf</i> , *). <i>a</i> contains the n -by- n matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. <i>af</i> contains the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> as computed by ?hetrf. The second dimension of <i>af</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of array <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of array <i>af</i> ; $ldaf \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION n . Details of the interchanges and the block structure of <i>D</i> as determined by ?hetrf.
<i>work</i>	REAL for cla_herpgrw DOUBLE PRECISION for zla_herpgrw. Array, DIMENSION $2*n$. Workspace.

See Also

?hetrf

?la_lin_berr

Computes component-wise relative backward error.

Syntax

FORTRAN 77:

```
call sla_lin_berr(n, nz, nrhs, res, ayb, berr)
call dla_lin_berr(n, nz, nrhs, res, ayb, berr)
call cla_lin_berr(n, nz, nrhs, res, ayb, berr)
call zla_lin_berr(n, nz, nrhs, res, ayb, berr)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?la_lin_berr computes a component-wise relative backward error from the formula:

$$\max(i) (\text{abs}(R(i)) / (\text{abs}(\text{op}(A_s)) * \text{abs}(Y) + \text{abs}(B_s))) (i)$$

where $\text{abs}(Z)$ is the component-wise value of the matrix or vector *Z*.

Input Parameters

<i>n</i>	INTEGER. The number of linear equations, the order of the matrix A ; $n \geq 0$.
<i>nz</i>	INTEGER. The parameter for guarding against spuriously zero residuals. $(nz+1) * \text{slamch}(\text{'Safe minimum'})$ is added to $R(i)$ in the numerator of the relative backward error formula. The default value is n .
<i>nrhs</i>	INTEGER. Number of right-hand sides, the number of columns in the matrices AYB , RES , and $BERR$; $nrhs \geq 0$.
<i>res, ayb</i>	REAL for <code>sla_lin_berr</code> , <code>cla_lin_berr</code> DOUBLE PRECISION for <code>dla_lin_berr</code> , <code>zla_lin_berr</code> Arrays, DIMENSION ($n, nrhs$). <i>res</i> is the residual matrix, that is, the matrix R in the relative backward error formula. <i>ayb</i> is the denominator of that formula, that is, the matrix $\text{abs}(op(A_s)) * \text{abs}(Y) + \text{abs}(B_s)$. The matrices A , Y , and B are from iterative refinement. See description of <code>?la_gerfsx_extended</code> .

Output Parameters

<i>berr</i>	REAL for <code>sla_lin_berr</code> DOUBLE PRECISION for <code>dla_lin_berr</code> COMPLEX for <code>cla_lin_berr</code> DOUBLE COMPLEX for <code>zla_lin_berr</code> The component-wise relative backward error.
-------------	--

See Also

[?lamch](#)
[?la_gerfsx_extended](#)

?la_porcond

Estimates the Skeel condition number for a symmetric positive-definite matrix.

Syntax

FORTRAN 77:

```
call sla_porcond( uplo, n, a, lda, af, ldaf, cmode, c, info, work, iwork )
call dla_porcond( uplo, n, a, lda, af, ldaf, cmode, c, info, work, iwork )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The function estimates the Skeel condition number of

`op(A) * op2(C)`

where

the `cmode` parameter determines `op2` as follows:

<code>cmode Value</code>	<code>op2(C)</code>
1	C
0	I
-1	$\text{inv}(C)$

The Skeel condition number

`cond(A) = norminf(|inv(A)| |A|)`

is computed by computing scaling factors R such that

`diag(R)*A*op2(C)`

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '. Specifies the triangle of A to store: If <code>uplo</code> = ' <code>U</code> ', the upper triangle of A is stored, If <code>uplo</code> = ' <code>L</code> ', the lower triangle of A is stored.
<code>n</code>	INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.
<code>a, af, c, work</code>	REAL for <code>sla_porcond</code> DOUBLE PRECISION for <code>dla_porcond</code> Arrays: <code>a (lda,*)</code> contains the n -by- n matrix A . <code>af (ldaf,*)</code> contains the triangular factor <code>L</code> or <code>U</code> from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$, as computed by <code>?potrf</code> . <code>c, DIMENSION n</code> . The vector C in the formula <code>op(A) * op2(C)</code> . <code>work</code> is a workspace array of DIMENSION $(3*n)$. The second dimension of <code>a</code> and <code>af</code> must be at least <code>max(1, n)</code> .
<code>lda</code>	INTEGER. The leading dimension of the array <code>ab</code> . $lda \geq \max(1, n)$.
<code>ldaf</code>	INTEGER. The leading dimension of <code>af</code> . $ldaf \geq \max(1, n)$.
<code>cmode</code>	INTEGER. Determines <code>op2(C)</code> in the formula <code>op(A) * op2(C)</code> as follows: If <code>cmode</code> = 1, <code>op2(C) = C</code> . If <code>cmode</code> = 0, <code>op2(C) = I</code> .

If $cmode = -1$, $\text{op2}(C) = \text{inv}(C)$.
 $iwork$ INTEGER. Workspace array with DIMENSION n .

Output Parameters

$info$ INTEGER.
If $info = 0$, the execution is successful.
If $i > 0$, the i -th parameter is invalid.

See Also

?potrf

?la_porcond_c

Computes the infinity norm condition number of
 $\text{op}(A)^*\text{inv}(\text{diag}(c))$ for Hermitian positive-definite
matrices.

Syntax

FORTRAN 77:

```
call cla_porcond_c( uplo, n, a, lda, af, ldaf, c, capply, info, work, rwork )
call zla_porcond_c( uplo, n, a, lda, af, ldaf, c, capply, info, work, rwork )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function computes the infinity norm condition number of

$\text{op}(A) * \text{inv}(\text{diag}(c))$

where the c is a REAL vector for `cla_porcond_c` and a DOUBLE PRECISION vector for `zla_porcond_c`.

Input Parameters

$uplo$ CHARACTER*1. Must be 'U' or 'L'.
Specifies the triangle of A to store:
If $uplo = 'U'$, the upper triangle of A is stored,
If $uplo = 'L'$, the lower triangle of A is stored.

n INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.

a COMPLEX for `cla_porcond_c`
DOUBLE COMPLEX for `zla_porcond_c`
Array, DIMENSION ($lda, *$). On entry, the n -by- n matrix A . The second dimension of a must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of the array *a*. $lda \geq \max(1, n)$.

af COMPLEX for `cla_porcond_c`
 DOUBLE COMPLEX for `zla_porcond_c`
 Array, DIMENSION (*ldaf*, *). The triangular factor L or U from the Cholesky factorization
 $A = U^H * U$ or $A = L * L^H$,
 as computed by `?potrf`.
 The second dimension of *af* must be at least $\max(1, n)$.

ldaf INTEGER. The leading dimension of the array *af*. $ldaf \geq \max(1, n)$.

c REAL for `cla_porcond_c`
 DOUBLE PRECISION for `zla_porcond_c`
 Array *c* with DIMENSION *n*. The vector *c* in the formula
 $op(A) * \text{inv}(\text{diag}(c))$.

capply LOGICAL. If .TRUE., then the function uses the vector *c* from the formula
 $op(A) * \text{inv}(\text{diag}(c))$.

work COMPLEX for `cla_porcond_c`
 DOUBLE COMPLEX for `zla_porcond_c`
 Array DIMENSION $2*n$. Workspace.

rwork REAL for `cla_porcond_c`
 DOUBLE PRECISION for `zla_porcond_c`
 Array DIMENSION *n*. Workspace.

Output Parameters

info INTEGER.
 If *info* = 0, the execution is successful.
 If *i* > 0, the *i*-th parameter is invalid.

See Also

`?potrf`

`?la_porcond_x`

Computes the infinity norm condition number of
 $op(A)*\text{diag}(x)$ for Hermitian positive-definite matrices.

Syntax

FORTRAN 77:

```
call cla_porcond_x( uplo, n, a, lda, af, ldaf, x, info, work, rwork )
call zla_porcond_x( uplo, n, a, lda, af, ldaf, x, info, work, rwork )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function computes the infinity norm condition number of

$\text{op}(A) * \text{diag}(x)$

where the x is a COMPLEX vector for `cla_porcond_x` and a DOUBLE COMPLEX vector for `zla_porcond_x`.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If <code>uplo</code> = 'U', the upper triangle of A is stored, If <code>uplo</code> = 'L', the lower triangle of A is stored.
<code>n</code>	INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.
<code>a</code>	COMPLEX for <code>cla_porcond_c</code> DOUBLE COMPLEX for <code>zla_porcond_c</code> Array, DIMENSION (<code>lda, *</code>). On entry, the n -by- n matrix A . The second dimension of <code>a</code> must be at least $\max(1, n)$.
<code>lda</code>	INTEGER. The leading dimension of the array <code>a</code> . $\text{lda} \geq \max(1, n)$.
<code>af</code>	COMPLEX for <code>cla_porcond_c</code> DOUBLE COMPLEX for <code>zla_porcond_c</code> Array, DIMENSION (<code>ldaf, *</code>). The triangular factor L or U from the Cholesky factorization $A = U^H * U$ or $A = L * L^H$, as computed by <code>?potrf</code> . The second dimension of <code>af</code> must be at least $\max(1, n)$.
<code>ldaf</code>	INTEGER. The leading dimension of the array <code>af</code> . $\text{ldaf} \geq \max(1, n)$.
<code>x</code>	COMPLEX for <code>cla_porcond_c</code> DOUBLE COMPLEX for <code>zla_porcond_c</code> Array <code>x</code> with DIMENSION n . The vector x in the formula $\text{op}(A) * \text{inv}(\text{diag}(x))$.
<code>work</code>	COMPLEX for <code>cla_porcond_c</code> DOUBLE COMPLEX for <code>zla_porcond_c</code> Array DIMENSION $2*n$. Workspace.
<code>rwork</code>	REAL for <code>cla_porcond_c</code>

DOUBLE PRECISION for zla_porcond_c
Array DIMENSION n . Workspace.

Output Parameters

info INTEGER.
If $info = 0$, the execution is successful.
If $i > 0$, the i -th parameter is invalid.

See Also

[?potrf](#)

?la_porfsx_extended

Improves the computed solution to a system of linear equations for symmetric or Hermitian positive-definite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

FORTRAN 77:

```
call sla_porfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, colequ, c, b,
                           ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
                           rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )

call dla_porfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, colequ, c, b,
                           ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
                           rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )

call cla_porfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, colequ, c, b,
                           ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
                           rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )

call zla_porfsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, colequ, c, b,
                           ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
                           rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?la_porfsx_extended subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The ?herfsx routine calls ?la_porfsx_extended to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for *err_bnds_norm* and *err_bnds_comp* for details of the error bounds.

Use ?la_porfsx_extended to set only the second fields of *err_bnds_norm* and *err_bnds_comp*.

Input Parameters

<i>prec_type</i>	INTEGER. Specifies the intermediate precision to be used in refinement. The value is defined by <code>ilaprec(p)</code> , where <i>p</i> is a CHARACTER and: If <i>p</i> = 'S': Single. If <i>p</i> = 'D': Double. If <i>p</i> = 'I': Indigenous. If <i>p</i> = 'X', 'E': Extra.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of <i>A</i> to store: If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored, If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The number of linear equations; the order of the matrix <i>A</i> ; <i>n</i> ≥ 0.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the matrix <i>B</i> .
<i>a</i> , <i>af</i> , <i>b</i> , <i>y</i>	REAL for <code>sla_porfsx_extended</code> DOUBLE PRECISION for <code>dla_porfsx_extended</code> COMPLEX for <code>cla_porfsx_extended</code> DOUBLE COMPLEX for <code>zla_porfsx_extended</code> . Arrays: <i>a</i> (<i>lda</i> , *), <i>af</i> (<i>ldaf</i> , *), <i>b</i> (<i>ldb</i> , *), <i>y</i> (<i>ldy</i> , *). The array <i>a</i> contains the original <i>n</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least <code>max(1, n)</code> . The array <i>af</i> contains the triangular factor <i>L</i> or <i>U</i> from the Cholesky factorization as computed by <code>?potrf</code> : $A = U^T * U$ or $A = L * L^T$ for real flavors, $A = U^H * U$ or $A = L * L^H$ for complex flavors. The second dimension of <i>af</i> must be at least <code>max(1, n)</code> . The array <i>b</i> contains the right-hand-side of the matrix <i>B</i> . The second dimension of <i>b</i> must be at least <code>max(1, nrhs)</code> . The array <i>y</i> on entry contains the solution matrix <i>X</i> as computed by <code>?potrs</code> . The second dimension of <i>y</i> must be at least <code>max(1, nrhs)</code> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> ; <i>lda</i> ≥ <code>max(1, n)</code> .
<i>ldaf</i>	INTEGER. The leading dimension of the array <i>af</i> ; <i>ldaf</i> ≥ <code>max(1, n)</code> .
<i>colequ</i>	LOGICAL. If <i>colequ</i> = .TRUE., column equilibration was done to <i>A</i> before calling this routine. This is needed to compute the solution and error bounds correctly.
<i>c</i>	REAL for <code>sla_porfsx_extended</code> and <code>cla_porfsx_extended</code>

DOUBLE PRECISION for `dla_porfsx_extended` and
`zla_porfsx_extended`.

`c` contains the column scale factors for `A`. If `colequ = .FALSE.`, `c` is not used.

If `c` is input, each element of `c` should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

`ldb`

INTEGER. The leading dimension of the array `b`; $ldb \geq \max(1, n)$.

`ldy`

INTEGER. The leading dimension of the array `y`; $ldy \geq \max(1, n)$.

`n_norms`

INTEGER. Determines which error bounds to return. See `err_bnds_norm` and `err_bnds_comp` descriptions in *Output Arguments* section below.

If $n_norms \geq 1$, returns normwise error bounds.

If $n_norms \geq 2$, returns componentwise error bounds.

`err_bnds_norm`

REAL for `sla_porfsx_extended` and `cla_porfsx_extended`
 DOUBLE PRECISION for `dla_porfsx_extended` and
`zla_porfsx_extended`.

Array, DIMENSION (`nrhs, n_err_bnds`). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error.

Normwise relative error in the i -th solution vector is defined as follows:

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i,:)` corresponds to the i -th right-hand side.

The second index in `err_bnds_norm(:,err)` contains the following three fields:

`err=1`

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\text{sqrt}(n) * \text{slamch}(\varepsilon)$ for `sla_porfsx_extended/cla_porfsx_extended` and $\text{sqrt}(n) * \text{dlamch}(\varepsilon)$ for `dla_porfsx_extended/zla_porfsx_extended`.

`err=2`

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is

greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$
 for
 $\text{sla_porfsx_extended}/\text{cla_porfsx_extended}$
and $\sqrt{n} * \text{dlamch}(\epsilon)$ for
 $\text{dla_porfsx_extended}/\text{zla_porfsx_extended}$. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated normwise reciprocal condition number.
 Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for
 $\text{sla_porfsx_extended}/\text{cla_porfsx_extended}$
and $\sqrt{n} * \text{dlamch}(\epsilon)$ for
 $\text{dla_porfsx_extended}/\text{zla_porfsx_extended}$ to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix Z .

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

`err_bnds_comp`

REAL for $\text{sla_porfsx_extended}$ and $\text{cla_porfsx_extended}$
 DOUBLE PRECISION for $\text{dla_porfsx_extended}$ and $\text{zla_porfsx_extended}$.

Array, DIMENSION ($\text{nrhs}, n_{\text{err_bnds}}$). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{\text{true}}_{ji} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params}(3) = 0.0$), then `err_bnds_comp` is not accessed. If $n_{\text{err_bnds}} < 3$, then at most the first $(:, n_{\text{err_bnds}})$ entries are returned.

The first index in `err_bnds_comp(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:, err)` contains the following three fields:

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>sla_porfsx_extended/cla_porfsx_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>dla_porfsx_extended/zla_porfsx_extended</code> .
<code>err=2</code>	"Guaranteed" error bbound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>sla_porfsx_extended/cla_porfsx_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>dla_porfsx_extended/zla_porfsx_extended</code> . This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for <code>sla_porfsx_extended/cla_porfsx_extended</code> and $\sqrt{n} * \text{dlamch}(\epsilon)$ for <code>dla_porfsx_extended/zla_porfsx_extended</code> to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix Z . Let $z = s^* (a^* \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a^* \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1. Use this subroutine to set only the second field above.

```
res, dy, y_tail
REAL for sla_porfsx_extended
DOUBLE PRECISION for dla_porfsx_extended
COMPLEX for cla_porfsx_extended
DOUBLE COMPLEX for zla_porfsx_extended.

Workspace arrays of DIMENSION n.
res holds the intermediate residual.
dy holds the intermediate solution.
y_tail holds the trailing bits of the intermediate solution.
```

`ayb` REAL for `sla_porfsx_extended` and `cla_porfsx_extended`

DOUBLE PRECISION for `dla_porfsx_extended` and
`zla_porfsx_extended`.

Workspace array, DIMENSION n .

rcond REAL for `sla_porfsx_extended` and `cla_porfsx_extended`

DOUBLE PRECISION for `dla_porfsx_extended` and
`zla_porfsx_extended`.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If *rcond* is less than the machine precision, in particular, if *rcond* = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

ithresh INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

rthresh REAL for `sla_porfsx_extended` and `cla_porfsx_extended`

DOUBLE PRECISION for `dla_porfsx_extended` and
`zla_porfsx_extended`.

Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies

$\text{norm}(\text{dx}_{\{i+1\}}) < rthresh * \text{norm}(\text{dx}_i)$

where $\text{norm}(z)$ is the infinity norm of Z .

rthresh satisfies

$0 < rthresh \leq 1$.

The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.

dz_ub REAL for `sla_porfsx_extended` and `cla_porfsx_extended`

DOUBLE PRECISION for `dla_porfsx_extended` and
`zla_porfsx_extended`.

Determines when to start considering componentwise convergence. Componentwise *dz_ub* convergence is only considered after each component of the solution y is stable, that is, the relative change in each component is less than *dz_ub*. The default value is 0.25, requiring the first bit to be stable.

ignore_cwise LOGICAL

If .TRUE., the function ignores componentwise convergence. Default value is .FALSE..

Output Parameters

y REAL for `sla_porfsx_extended`

DOUBLE PRECISION for `dla_porfsx_extended`
 COMPLEX for `cla_porfsx_extended`
 DOUBLE COMPLEX for `zla_porfsx_extended`.

The improved solution matrix Y .

`berr_out`

REAL for `sla_porfsx_extended` and `cla_porfsx_extended`
 DOUBLE PRECISION for `dla_porfsx_extended` and
`zla_porfsx_extended`.

Array, DIMENSION $nrhs$. `berr_out(j)` contains the componentwise relative backward error for right-hand-side j from the formula

$$\max(i) \left(\frac{\text{abs}(\text{res}(i))}{\left(\text{abs}(\text{op}(A)) * \text{abs}(y) + \text{abs}(B) \right)(i)} \right)$$

where `abs(z)` is the componentwise absolute value of the matrix or vector Z . This is computed by `?la_lin_berr`.

`err_bnds_norm`,
`err_bnds_comp`

Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array ($1:nrhs, 2$). The other elements are kept unchanged.

`info`

INTEGER. If $info = 0$, the execution is successful. The solution to every right-hand side is guaranteed.
 If $info = -i$, the i -th parameter had an illegal value.

See Also

[?porfsx](#)
[?potrf](#)
[?potrs](#)
[?lamch](#)
[ilaprec](#)
[ilatrans](#)
[?la_lin_berr](#)

?la_porpvgrw

Computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$ for a symmetric or Hermitian positive-definite matrix.

Syntax

FORTRAN 77:

```
call sla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
call dla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
call cla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
call zla_porpvgrw( uplo, ncols, a, lda, af, ldaf, work )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The `?la_porpvgrw` routine computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element norm* is used. If this is much less than 1, the stability of the *LU* factorization of the equilibrated matrix A could be poor. This also means that the solution X , estimated condition numbers, and error bounds could be unreliable.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '.
	Specifies the triangle of A to store: If <code>uplo</code> = ' <code>U</code> ', the upper triangle of A is stored, If <code>uplo</code> = ' <code>L</code> ', the lower triangle of A is stored.
<code>ncols</code>	INTEGER. The number of columns of the matrix A ; $ncols \geq 0$.
<code>a</code> , <code>af</code>	REAL for <code>sla_porpvgrw</code> DOUBLE PRECISION for <code>dla_porpvgrw</code> COMPLEX for <code>cla_porpvgrw</code> DOUBLE COMPLEX for <code>zla_porpvgrw</code> . Arrays: $a(1:lda, *)$, $af(1:ldaf, *)$. The array <code>a</code> contains the input n -by- n matrix A . The second dimension of <code>a</code> must be at least $\max(1, n)$. The array <code>af</code> contains the triangular factor <code>L</code> or <code>U</code> from the Cholesky factorization as computed by <code>?potrf</code> : $A = U^T * U$ or $A = L * L^T$ for real flavors, $A = U^H * U$ or $A = L * L^H$ for complex flavors. The second dimension of <code>af</code> must be at least $\max(1, n)$.
<code>lda</code>	INTEGER. The leading dimension of <code>a</code> ; $lda \geq \max(1, n)$.
<code>ldaf</code>	INTEGER. The leading dimension of <code>af</code> ; $ldaf \geq \max(1, n)$.
<code>work</code>	REAL for <code>sla_porpvgrw</code> and <code>cla_porpvgrw</code> DOUBLE PRECISION for <code>dla_porpvgrw</code> and <code>zla_porpvgrw</code> . Workspace array, dimension $2*n$.

See Also

`?potrf`

`?laqhe`

Scales a Hermitian matrix.

Syntax

```
call claqhe( uplo, n, a, lda, s, scond, amax, equed )
call zlaqhe( uplo, n, a, lda, s, scond, amax, equed )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine equilibrates a Hermitian matrix A using the scaling factors in the vector s .

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether to store the upper or lower part of the Hermitian matrix A . If <i>uplo</i> = 'U', the upper triangular part of A ; if <i>uplo</i> = 'L', the lower triangular part of A .
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$.
<i>a</i>	COMPLEX for claqhe DOUBLE COMPLEX for zlaghe Array, DIMENSION (<i>lda</i> , <i>n</i>). On entry, the Hermitian matrix A . If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of <i>a</i> contains the upper triangular part of matrix A and the strictly lower triangular part of <i>a</i> is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of <i>a</i> contains the lower triangular part of matrix A and the strictly upper triangular part of <i>a</i> is not referenced.
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> . <i>lda</i> $\geq \max(n, 1)$.
<i>s</i>	REAL for claqhe DOUBLE PRECISION for zlaghe Array, DIMENSION (<i>n</i>). The scale factors for A .
<i>scond</i>	REAL for claqhe DOUBLE PRECISION for zlaghe Ratio of the smallest $s(i)$ to the largest $s(i)$.
<i>amax</i>	REAL for claqhe DOUBLE PRECISION for zlaghe Absolute value of largest matrix entry.

Output Parameters

<i>a</i>	If <i>equed</i> = 'Y', <i>a</i> contains the equilibrated matrix $\text{diag}(s) * A * \text{diag}(s)$.
----------	--

equed CHARACTER*1.

Specifies whether or not equilibration was done.

If `eqed = 'N'`: No equilibration.

If `equed = 'Y'`: Equilibration was done, that is, A has been replaced by $\text{diag}(s)^*A^*\text{diag}(s)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*. The parameter *thresh* is a threshold value used to decide if scaling should be done based on the ratio of the scaling factors. If $scond < thresh$, scaling is done.

The `large` and `small` parameters are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If `amax > large` or `amax < small`, scaling is done.

?laghp

Scales a Hermitian matrix stored in packed form.

Syntax

```
call claqhp( uplo, n, ap, s, scond, amax, equed )
call zlaqhp( uplo, n, ap, s, scond, amax, equed )
```

Include Files

- Fortran: mkl.fi
 - C: mkl.h

Description

The routine equilibrates a Hermitian matrix A using the scaling factors in the vector s .

Input Parameters

uplo CHARACTER*1.

Specifies whether to store the upper or lower part of the Hermitian matrix A .

If $\text{uplo} = \text{'U'}$, the upper triangular part of A ;

if $uplo = 'L'$, the lower triangular part of A .

$$n \geq 0.$$

ap COMPLEX for claghp

DOUBLE COMPLEX for zlaqhp

Array, DIMENSION ($n*(n+1)/2$). The Hermitian matrix A.

- If $uplo = 'U'$, the upper triangular part of the Hermitian matrix A is stored in the packed array ap as follows:

$$ap(i + (\lceil j-1 \rceil) * \lceil j/2 \rceil) = A(i, j) \text{ for } 1 \leq i \leq j.$$

- If $uplo = 'L'$, the lower triangular part of Hermitian matrix A is stored in the packed array ap as follows:

$$ap(i + (j-1)*(2n-j)/2) = A(i, j) \text{ for } j \leq i \leq n.$$

*s*REAL for `claqhp`DOUBLE PRECISION for `zlaqhp`Array, DIMENSION (*n*). The scale factors for A .*scond*REAL for `claqhp`DOUBLE PRECISION for `zlaqhp`Ratio of the smallest $s(i)$ to the largest $s(i)$.*amax*REAL for `claqhp`DOUBLE PRECISION for `zlaqhp`

Absolute value of largest matrix entry.

Output Parameters

*a*If $equed = 'Y'$, *a* contains the equilibrated matrix $\text{diag}(s) * A * \text{diag}(s)$ in the same storage format as on input.*equed*

CHARACTER*1.

Specifies whether or not equilibration was done.

If $equed = 'N'$: No equilibration.If $equed = 'Y'$: Equilibration was done, that is, *A* has been replaced by $\text{diag}(s) * A * \text{diag}(s)$.

Application Notes

The routine uses internal parameters *thresh*, *large*, and *small*. The parameter *thresh* is a threshold value used to decide if scaling should be done based on the ratio of the scaling factors. If $scond < \text{thresh}$, scaling is done.

The *large* and *small* parameters are threshold values used to decide if scaling should be done based on the absolute size of the largest matrix element. If $amax > \text{large}$ or $amax < \text{small}$, scaling is done.

?larcm

Multiples a square real matrix by a complex matrix.

Syntax

```
call clarcm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

```
call zlarcm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine performs a simple matrix-matrix multiplication of the form

$$C = A^*B,$$

where A is m -by- m and real, B is m -by- n and complex, C is m -by- n and complex.

Input Parameters

m	INTEGER. The number of rows and columns of the matrix A and of the number of rows of the matrix C ($m \geq 0$).
n	INTEGER. The number of columns of the matrix B and the number of columns of the matrix C ($n \geq 0$).
a	REAL for clarcm DOUBLE PRECISION for zlarcm Array, DIMENSION (lda, m). Contains the m -by- m matrix A .
lda	INTEGER. The leading dimension of the array a , $lda \geq \max(1, m)$.
b	COMPLEX for clarcm DOUBLE COMPLEX for zlarcm Array, DIMENSION (ldb, n). Contains the m -by- n matrix B .
ldb	INTEGER. The leading dimension of the array b , $ldb \geq \max(1, n)$.
ldc	INTEGER. The leading dimension of the output array c , $ldc \geq \max(1, m)$.
$rwork$	REAL for clarcm DOUBLE PRECISION for zlarcm Workspace array, DIMENSION ($2*m*n$).

Output Parameters

c	COMPLEX for clarcm DOUBLE COMPLEX for zlarcm Array, DIMENSION (ldc, n). Contains the m -by- n matrix C .
-----	--

?la_gerpvgew

Computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$ for a general matrix.

Syntax

FORTRAN 77:

```
call sla_gerpvgew( n, ncols, a, lda, af, ldaf )
call dla_gerpvgew( n, ncols, a, lda, af, ldaf )
```

```
call cla_gerpvgrw( n, ncols, a, lda, af, ldaf )
call zla_gerpvgrw( n, ncols, a, lda, af, ldaf )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?la_gerpvgrw routine computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element* norm is used. If this is much less than 1, the stability of the LU factorization of the equilibrated matrix A could be poor. This also means that the solution X, estimated condition numbers, and error bounds could be unreliable.

Input Parameters

<i>n</i>	INTEGER. The number of linear equations, the order of the matrix <i>A</i> ; $n \geq 0$.
<i>ncols</i>	INTEGER. The number of columns of the matrix <i>A</i> ; $ncols \geq 0$.
<i>a</i> , <i>af</i>	REAL for sla_gerpvgrw DOUBLE PRECISION for dla_gerpvgrw COMPLEX for cla_gerpvgrw DOUBLE COMPLEX for zla_gerpvgrw. Arrays: <i>a</i> (<i>lda</i> , *), <i>af</i> (<i>ldaf</i> , *).
	The array <i>a</i> contains the input <i>n</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least $\max(1, n)$. The array <i>af</i> contains the factors L and U from the factorization triangular factor L or U from the Cholesky factorization $A = P^*L^*U$ as computed by ?getrf. The second dimension of <i>af</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $lda \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of <i>af</i> ; $ldaf \geq \max(1, n)$.

See Also

?getrf

?larscl2

Performs reciprocal diagonal scaling on a vector.

Syntax

FORTRAN 77:

```
call slarscl2(m, n, d, x, ldx)
call dlarscl2(m, n, d, x, ldx)
call clarscl2(m, n, d, x, ldx)
call zlarscl2(m, n, d, x, ldx)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?larscl2 routines perform reciprocal diagonal scaling on a vector

$$x := D^{-1} \cdot x,$$

where:

x is a vector, and

D is a diagonal matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix D and the number of elements of the vector x . The value of m must be at least zero.
n	INTEGER. The number of columns of D and x . The value of n must be at least zero.
d	REAL for slarscl2 and clarscl2. DOUBLE PRECISION for dlarscl2 and zlarscl2. Array, DIMENSION m . Diagonal matrix D stored as a vector of length m .
x	REAL for slarscl2. DOUBLE PRECISION for dlarscl2. COMPLEX for clarscl2. DOUBLE COMPLEX for zlarscl2. Array, DIMENSION ($l dx, n$). The vector x to scale by D .
$l dx$	INTEGER. The leading dimension of the vector x . The value of $l dx$ must be at least zero.

Output Parameters

x	Scaled vector x .
-----	---------------------

?lascl2

Performs diagonal scaling on a vector.

Syntax

FORTRAN 77:

```
call slascl2(m, n, d, x, ldx)
call dlascl2(m, n, d, x, ldx)
call clascl2(m, n, d, x, ldx)
call zlascl2(m, n, d, x, ldx)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?lascl2 routines perform diagonal scaling on a vector

$x := D \cdot x$,

where:

x is a vector, and

D is a diagonal matrix.

Input Parameters

m	INTEGER. Specifies the number of rows of the matrix D and the number of elements of the vector x . The value of m must be at least zero.
n	INTEGER. The number of columns of D and x . The value of n must be at least zero.
d	REAL for slascl2 and clascl2. DOUBLE PRECISION for dlascl2 and zlascl2. Array, DIMENSION m . Diagonal matrix D stored as a vector of length m .
x	REAL for slascl2. DOUBLE PRECISION for dlascl2. COMPLEX for clascl2. DOUBLE COMPLEX for zlascl2. Array, DIMENSION ($l dx, n$). The vector x to scale by D .
$l dx$	INTEGER. The leading dimension of the vector x . The value of $l dx$ must be at least zero.

Output Parameters

x	Scaled vector x .
-----	---------------------

?la_syamv

Computes a matrix-vector product using a symmetric indefinite matrix to calculate error bounds.

Syntax

FORTRAN 77:

```
call sla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call dla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
call cla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
```

```
call zla_syamv(uplo, n, alpha, a, lda, x, incx, beta, y, incy)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?la_syamv routines perform a matrix-vector operation defined as

$$y := \alpha * \text{abs}(A) * \text{abs}(x) + \beta * \text{abs}(y),$$

where:

alpha and *beta* are scalars,

x and *y* are vectors,

A is an *n*-by-*n* Hermitian matrix.

This function is primarily used in calculating error bounds. To protect against underflow during evaluation, the function perturbs components in the resulting vector away from zero by $(n + 1)$ times the underflow threshold. To prevent unnecessarily large errors for block structure embedded in general matrices, the function does not perturb *symbolically* zero components. A zero entry is considered *symbolic* if all multiplications involved in computing that entry have at least one zero multiplicand.

Input Parameters

uplo

CHARACTER*1.

Specifies whether the upper or lower triangular part of the array *A* is to be referenced:

If *uplo* = 'BLAS_UPPER', only the upper triangular part of *A* is to be referenced,

If *uplo* = 'BLAS_LOWER', only the lower triangular part of *A* is to be referenced.

n

INTEGER. Specifies the number of rows and columns of the matrix *A*. The value of *n* must be at least zero.

alpha

REAL for *sla_syamv* and *cla_syamv*

DOUBLE PRECISION for *dla_syamv* and *zla_syamv*.

Specifies the scalar *alpha*.

a

REAL for *sla_syamv*

DOUBLE PRECISION for *dla_syamv*

COMPLEX for *cla_syamv*

DOUBLE COMPLEX for *zla_syamv*.

Array, DIMENSION (*lda*, *). Before entry, the leading *m*-by-*n* part of the array *a* must contain the matrix of coefficients. The second dimension of *a* must be at least $\max(1, n)$.

lda

INTEGER. Specifies the leading dimension of *a* as declared in the calling (sub)program. The value of *lda* must be at least $\max(1, n)$.

x

REAL for *sla_syamv*

DOUBLE PRECISION for `dla_syamv`
 COMPLEX for `cla_syamv`

DOUBLE COMPLEX for `zla_syamv`.

Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incx))$. Before entry, the incremented array x must contain the vector X .

incx INTEGER. Specifies the increment for the elements of x .

The value of *incx* must be non-zero.

beta REAL for `sla_syamv` and `cla_syamv`

DOUBLE PRECISION for `dla_syamv` and `zla_syamv`

Specifies the scalar *beta*. When *beta* is zero, you do not need to set y on input.

y REAL for `sla_syamv` and `cla_syamv`

DOUBLE PRECISION for `dla_syamv` and `zla_syamv`

Array, DIMENSION at least $(1 + (n - 1) * \text{abs}(incy))$ otherwise. Before entry with non-zero *beta*, the incremented array y must contain the vector Y .

incy INTEGER. Specifies the increment for the elements of y .

The value of *incy* must be non-zero.

Output Parameters

y Updated vector Y .

?la_syrcond

Estimates the Skeel condition number for a symmetric indefinite matrix.

Syntax

FORTRAN 77:

```
call sla_syrcond( uplo, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
call dla_syrcond( uplo, n, a, lda, af, ldaf, ipiv, cmode, c, info, work, iwork )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The function estimates the Skeel condition number of

$\text{op}(A) * \text{op}2(C)$

where

the *cmode* parameter determines $\text{op}2$ as follows:

<i>cmode</i> Value	<i>op2(C)</i>
1	C
0	I
-1	$\text{inv}(C)$

The Skeel condition number

`cond(A) = norminf(|inv(A)| |A|)`

is computed by computing scaling factors R such that

`diag(R) * A * op2(C)`

is row equilibrated and by computing the standard infinity-norm condition number.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If <i>uplo</i> = 'U', the upper triangle of A is stored, If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.
<i>a, af, c, work</i>	REAL for <code>sla_syrcond</code> DOUBLE PRECISION for <code>dla_syrcond</code> Arrays: <i>ab</i> (<i>lda,*</i>) contains the n -by- n matrix A . <i>af</i> (<i>ldaf,*</i>) contains the The block diagonal matrix D and the multipliers used to obtain the factor L or U as computed by <code>?sytrf</code> . The second dimension of <i>a</i> and <i>af</i> must be at least $\max(1, n)$. <i>c</i> , DIMENSION n . The vector C in the formula $\text{op}(A) * \text{op2}(C)$. <i>work</i> is a workspace array of DIMENSION $(3*n)$.
<i>lda</i>	INTEGER. The leading dimension of the array <i>ab</i> . $\text{lda} \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of <i>af</i> . $\text{ldaf} \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array with DIMENSION n . Details of the interchanges and the block structure of D as determined by <code>?sytrf</code> .
<i>cmode</i>	INTEGER. Determines <i>op2(C)</i> in the formula $\text{op}(A) * \text{op2}(C)$ as follows: If <i>cmode</i> = 1, $\text{op2}(C) = C$. If <i>cmode</i> = 0, $\text{op2}(C) = I$. If <i>cmode</i> = -1, $\text{op2}(C) = \text{inv}(C)$.

iwork INTEGER. Workspace array with DIMENSION n .

Output Parameters

info INTEGER.

If $info = 0$, the execution is successful.

If $i > 0$, the i -th parameter is invalid.

See Also

[?sytrf](#)

?la_syrcond_c

Computes the infinity norm condition number of $\text{op}(A) * \text{inv}(\text{diag}(c))$ for symmetric indefinite matrices.

Syntax

FORTRAN 77:

```
call cla_syrcond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
call zla_syrcond_c( uplo, n, a, lda, af, ldaf, ipiv, c, capply, info, work, rwork )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function computes the infinity norm condition number of

$\text{op}(A) * \text{inv}(\text{diag}(c))$

where the c is a REAL vector for `cla_syrcond_c` and a DOUBLE PRECISION vector for `zla_syrcond_c`.

Input Parameters

uplo CHARACTER*1. Must be 'U' or 'L'.

Specifies the triangle of A to store:

If $uplo = 'U'$, the upper triangle of A is stored,

If $uplo = 'L'$, the lower triangle of A is stored.

n INTEGER. The number of linear equations, that is, the order of the matrix A ; $n \geq 0$.

a COMPLEX for `cla_syrcond_c`

DOUBLE COMPLEX for `zla_syrcond_c`

Array, DIMENSION ($lda, *$). On entry, the n -by- n matrix A . The second dimension of a must be at least $\max(1, n)$.

lda INTEGER. The leading dimension of the array a . $lda \geq \max(1, n)$.

af COMPLEX for `cla_syrcond_c`

DOUBLE COMPLEX for zla_syrcond_c
 Array, DIMENSION (*ldaf*, *). The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?sytrf. The second dimension of *af* must be at least $\max(1, n)$.

ldaf INTEGER. The leading dimension of the array *af*. $ldaf \geq \max(1, n)$.

ipiv INTEGER.
 Array with DIMENSION *n*. Details of the interchanges and the block structure of D as determined by ?sytrf.

c
 REAL for cla_syrcond_c
 DOUBLE PRECISION for zla_syrcond_c
 Array *c* with DIMENSION *n*. The vector *c* in the formula
 $op(A) * \text{inv}(\text{diag}(c))$.

capply LOGICAL. If .TRUE., then the function uses the vector *c* from the formula
 $op(A) * \text{inv}(\text{diag}(c))$.

work COMPLEX for cla_syrcond_c
 DOUBLE COMPLEX for zla_syrcond_c
 Array DIMENSION $2*n$. Workspace.

rwork REAL for cla_syrcond_c
 DOUBLE PRECISION for zla_syrcond_c
 Array DIMENSION *n*. Workspace.

Output Parameters

info INTEGER.
 If *info* = 0, the execution is successful.
 If *i* > 0, the *i*-th parameter is invalid.

See Also

?sytrf

?la_syrcond_x

Computes the infinity norm condition number of $op(A)*\text{diag}(x)$ for symmetric indefinite matrices.

Syntax

FORTRAN 77:

```
call cla_syrcond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
call zla_syrcond_x( uplo, n, a, lda, af, ldaf, ipiv, x, info, work, rwork )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function computes the infinity norm condition number of

$\text{op}(A) * \text{diag}(x)$

where the x is a COMPLEX vector for `cla_syrcond_x` and a DOUBLE COMPLEX vector for `zla_syrcond_x`.

Input Parameters

<code>uplo</code>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If <code>uplo</code> = 'U', the upper triangle of A is stored, If <code>uplo</code> = 'L', the lower triangle of A is stored.
<code>n</code>	INTEGER. The number of linear equations, that is, the order of the matrix A; $n \geq 0$.
<code>a</code>	COMPLEX for <code>cla_syrcond_c</code> DOUBLE COMPLEX for <code>zla_syrcond_c</code> Array, DIMENSION (<code>lda, *</code>). On entry, the n -by- n matrix A. The second dimension of <code>a</code> must be at least $\max(1, n)$.
<code>lda</code>	INTEGER. The leading dimension of the array <code>a</code> . $\text{lda} \geq \max(1, n)$.
<code>af</code>	COMPLEX for <code>cla_syrcond_c</code> DOUBLE COMPLEX for <code>zla_syrcond_c</code> Array, DIMENSION (<code>ldaf, *</code>). The block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by <code>?sytrf</code> . The second dimension of <code>af</code> must be at least $\max(1, n)$.
<code>ldaf</code>	INTEGER. The leading dimension of the array <code>af</code> . $\text{ldaf} \geq \max(1, n)$.
<code>ipiv</code>	INTEGER. Array with DIMENSION n . Details of the interchanges and the block structure of D as determined by <code>?sytrf</code> .
<code>x</code>	COMPLEX for <code>cla_syrcond_c</code> DOUBLE COMPLEX for <code>zla_syrcond_c</code> Array <code>x</code> with DIMENSION n . The vector x in the formula $\text{op}(A) * \text{inv}(\text{diag}(x))$.
<code>work</code>	COMPLEX for <code>cla_syrcond_c</code> DOUBLE COMPLEX for <code>zla_syrcond_c</code> Array DIMENSION $2*n$. Workspace.

rwork REAL for `cla_syrcond_c`
 DOUBLE PRECISION for `zla_syrcond_c`
 Array DIMENSION *n*. Workspace.

Output Parameters

info INTEGER.
 If *info* = 0, the execution is successful.
 If *i* > 0, the *i*-th parameter is invalid.

See Also

[?sytrf](#)

?la_syrfxsx_extended

Improves the computed solution to a system of linear equations for symmetric indefinite matrices by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution.

Syntax

FORTRAN 77:

```
call sla_syrfxsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )

call dla_syrfxsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )

call cla_syrfxsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )

call zla_syrfxsx_extended( prec_type, uplo, n, nrhs, a, lda, af, ldaf, ipiv, colequ, c,
b, ldb, y, ldy, berr_out, n_norms, err_bnds_norm, err_bnds_comp, res, ayb, dy, y_tail,
rcond, ithresh, rthresh, dz_ub, ignore_cwise, info )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The `?la_syrfxsx_extended` subroutine improves the computed solution to a system of linear equations by performing extra-precise iterative refinement and provides error bounds and backward error estimates for the solution. The `?syrfxsx` routine calls `?la_syrfxsx_extended` to perform iterative refinement.

In addition to normwise error bound, the code provides maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

Use `?la_syrfxsx_extended` to set only the second fields of `err_bnds_norm` and `err_bnds_comp`.

Input Parameters

<i>prec_type</i>	INTEGER. Specifies the intermediate precision to be used in refinement. The value is defined by <code>ilaprec(p)</code> , where <i>p</i> is a CHARACTER and: If <i>p</i> = 'S': Single. If <i>p</i> = 'D': Double. If <i>p</i> = 'I': Indigenous. If <i>p</i> = 'X', 'E': Extra.
<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of <i>A</i> to store: If <i>uplo</i> = 'U', the upper triangle of <i>A</i> is stored, If <i>uplo</i> = 'L', the lower triangle of <i>A</i> is stored.
<i>n</i>	INTEGER. The number of linear equations; the order of the matrix <i>A</i> ; <i>n</i> ≥ 0.
<i>nrhs</i>	INTEGER. The number of right-hand sides; the number of columns of the matrix <i>B</i> .
<i>a</i> , <i>af</i> , <i>b</i> , <i>y</i>	REAL for <code>sla_syrfsx_extended</code> DOUBLE PRECISION for <code>dla_syrfsx_extended</code> COMPLEX for <code>cla_syrfsx_extended</code> DOUBLE COMPLEX for <code>zla_syrfsx_extended</code> . Arrays: <i>a</i> (<i>lda</i> , *), <i>af</i> (<i>ldaf</i> , *), <i>b</i> (<i>ldb</i> , *), <i>y</i> (<i>ldy</i> , *). The array <i>a</i> contains the original <i>n</i> -by- <i>n</i> matrix <i>A</i> . The second dimension of <i>a</i> must be at least <code>max(1, n)</code> . The array <i>af</i> contains the block diagonal matrix <i>D</i> and the multipliers used to obtain the factor <i>U</i> or <i>L</i> as computed by <code>?sytrf</code> . The second dimension of <i>af</i> must be at least <code>max(1, n)</code> . The array <i>b</i> contains the right-hand-side of the matrix <i>B</i> . The second dimension of <i>b</i> must be at least <code>max(1, nrhs)</code> . The array <i>y</i> on entry contains the solution matrix <i>X</i> as computed by <code>?sytrs</code> . The second dimension of <i>y</i> must be at least <code>max(1, nrhs)</code> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> ; <i>lda</i> ≥ <code>max(1, n)</code> .
<i>ldaf</i>	INTEGER. The leading dimension of the array <i>af</i> ; <i>ldaf</i> ≥ <code>max(1, n)</code> .
<i>ipiv</i>	INTEGER. Array with DIMENSION <i>n</i> . Details of the interchanges and the block structure of <i>D</i> as determined by <code>?sytrf</code> .
<i>colequ</i>	LOGICAL. If <i>colequ</i> = .TRUE., column equilibration was done to <i>A</i> before calling this routine. This is needed to compute the solution and error bounds correctly.

c

REAL for `sla_syrfxsx_extended` and `cla_syrfxsx_extended`
 DOUBLE PRECISION for `dla_syrfxsx_extended` and
`zla_syrfxsx_extended`.

c contains the column scale factors for *A*. If *colequ* = .FALSE., *c* is not used.

If *c* is input, each element of *c* should be a power of the radix to ensure a reliable solution and error estimates. Scaling by power of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

ldb

INTEGER. The leading dimension of the array *b*; *ldb* $\geq \max(1, n)$.

ldy

INTEGER. The leading dimension of the array *y*; *ldy* $\geq \max(1, n)$.

n_norms

INTEGER. Determines which error bounds to return. See `err_bnds_norm` and `err_bnds_comp` descriptions in *Output Arguments* section below.

If *n_norms* ≥ 1 , returns normwise error bounds.

If *n_norms* ≥ 2 , returns componentwise error bounds.

err_bnds_norm

REAL for `sla_syrfxsx_extended` and `cla_syrfxsx_extended`
 DOUBLE PRECISION for `dla_syrfxsx_extended` and
`zla_syrfxsx_extended`.

Array, DIMENSION (*nrhs, n_err_bnds*). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error.

Normwise relative error in the *i*-th solution vector is defined as follows:

$$\frac{\max_j |X_{true_{ji}} - X_{ji}|}{\max_j |X_{ji}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

The first index in `err_bnds_norm(i,:)` corresponds to the *i*-th right-hand side.

The second index in `err_bnds_norm(:,err)` contains the following three fields:

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold <code>sqrt(n)*slamch('E')</code> for <code>sla_syrfxsx_extended/cla_syrfxsx_extended</code> and <code>sqrt(n)*dlamch('E')</code> for <code>dla_syrfxsx_extended/zla_syrfxsx_extended</code>
.	

`err=2`

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for
`sla_syrfsx_extended/cla_syrfsx_extended` and $\sqrt{n} * \text{dlamch}(\epsilon)$ for
`dla_syrfsx_extended/zla_syrfsx_extended`. This error bound should only be trusted if the previous boolean is true.

`err=3`

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for
`sla_syrfsx_extended/cla_syrfsx_extended` and $\sqrt{n} * \text{dlamch}(\epsilon)$ for
`dla_syrfsx_extended/zla_syrfsx_extended` to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \text{inf}) * \text{norm}(z, \text{inf}))$ for some appropriately scaled matrix Z .

Let $z = s * a$, where s scales each row by a power of the radix so all absolute row sums of z are approximately 1.

Use this subroutine to set only the second field above.

`err_bnds_comp` REAL for `sla_syrfsx_extended` and `cla_syrfsx_extended`

DOUBLE PRECISION for `dla_syrfsx_extended` and `zla_syrfsx_extended`.

Array, DIMENSION ($nrhs, n_{\text{err_bnds}}$). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the i -th solution vector:

$$\max_j \frac{|X_{true_{ji}} - X_{ji}|}{|X_{ji}|}$$

The array is indexed by the right-hand side i , on which the componentwise relative error depends, and by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params}(3) = 0.0$), then `err_bnds_comp` is not accessed. If $n_{\text{err_bnds}} < 3$, then at most the first $(:, n_{\text{err_bnds}})$ entries are returned.

The first index in `err_bnds_comp(i, :)` corresponds to the i -th right-hand side.

The second index in `err_bnds_comp(:,err)` contains the following three fields:

<code>err=1</code>	"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt(n) * \text{slamch}(\epsilon)$ for <code>sla_syrfssx_extended/cla_syrfssx_extended</code> and $\sqrt(n) * \text{dlamch}(\epsilon)$ for <code>dla_syrfssx_extended/zla_syrfssx_extended</code> .
<code>err=2</code>	"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt(n) * \text{slamch}(\epsilon)$ for <code>sla_syrfssx_extended/cla_syrfssx_extended</code> and $\sqrt(n) * \text{dlamch}(\epsilon)$ for <code>dla_syrfssx_extended/zla_syrfssx_extended</code> . This error bound should only be trusted if the previous boolean is true.
<code>err=3</code>	Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt(n) * \text{slamch}(\epsilon)$ for <code>sla_syrfssx_extended/cla_syrfssx_extended</code> and $\sqrt(n) * \text{dlamch}(\epsilon)$ for <code>dla_syrfssx_extended/zla_syrfssx_extended</code> to determine if the error estimate is "guaranteed". These reciprocal condition numbers are $1 / (\text{norm}(1/z, \infty) * \text{norm}(z, \infty))$ for some appropriately scaled matrix Z . Let $z = s * (a * \text{diag}(x))$, where x is the solution for the current right-hand side and s scales each row of $a * \text{diag}(x)$ by a power of the radix so all absolute row sums of z are approximately 1. Use this subroutine to set only the second field above.

```
res, dy, y_tail
REAL for sla_syrfssx_extended
DOUBLE PRECISION for dla_syrfssx_extended
COMPLEX for cla_syrfssx_extended
DOUBLE COMPLEX for zla_syrfssx_extended.

Workspace arrays of DIMENSION n.

res holds the intermediate residual.
dy holds the intermediate solution.
y_tail holds the trailing bits of the intermediate solution.
```

<i>ayb</i>	REAL for <code>sla_syrfsx_extended</code> and <code>cla_syrfsx_extended</code> DOUBLE PRECISION for <code>dla_syrfsx_extended</code> and <code>zla_syrfsx_extended</code> . Workspace array, DIMENSION <i>n</i> .
<i>rcond</i>	REAL for <code>sla_syrfsx_extended</code> and <code>cla_syrfsx_extended</code> DOUBLE PRECISION for <code>dla_syrfsx_extended</code> and <code>zla_syrfsx_extended</code> . Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix <i>A</i> after equilibration (if done). If <i>rcond</i> is less than the machine precision, in particular, if <i>rcond</i> = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.
<i>ithresh</i>	INTEGER. The maximum number of residual computations allowed for refinement. The default is 10. For 'aggressive', set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in <code>err_bnds_norm</code> and <code>err_bnds_comp</code> may no longer be trustworthy.
<i>rthresh</i>	REAL for <code>sla_syrfsx_extended</code> and <code>cla_syrfsx_extended</code> DOUBLE PRECISION for <code>dla_syrfsx_extended</code> and <code>zla_syrfsx_extended</code> . Determines when to stop refinement if the error estimate stops decreasing. Refinement stops when the next solution no longer satisfies $\text{norm}(\text{dx}_{\{i+1\}}) < \text{rthresh} * \text{norm}(\text{dx}_i)$ where $\text{norm}(z)$ is the infinity norm of <i>Z</i> . <i>rthresh</i> satisfies $0 < \text{rthresh} \leq 1$. The default value is 0.5. For 'aggressive' set to 0.9 to permit convergence on extremely ill-conditioned matrices.
<i>dz_ub</i>	REAL for <code>sla_syrfsx_extended</code> and <code>cla_syrfsx_extended</code> DOUBLE PRECISION for <code>dla_syrfsx_extended</code> and <code>zla_syrfsx_extended</code> . Determines when to start considering componentwise convergence. Componentwise <i>dz_ub</i> convergence is only considered after each component of the solution <i>y</i> is stable, that is, the relative change in each component is less than <i>dz_ub</i> . The default value is 0.25, requiring the first bit to be stable.
<i>ignore_cwise</i>	LOGICAL If .TRUE., the function ignores componentwise convergence. Default value is .FALSE..

Output Parameters

<i>y</i>	REAL for <code>sla_syrfxsx_extended</code> DOUBLE PRECISION for <code>dla_syrfxsx_extended</code> COMPLEX for <code>cla_syrfxsx_extended</code> DOUBLE COMPLEX for <code>zla_syrfxsx_extended</code> . The improved solution matrix <i>Y</i> .
<i>berr_out</i>	REAL for <code>sla_syrfxsx_extended</code> and <code>cla_syrfxsx_extended</code> DOUBLE PRECISION for <code>dla_syrfxsx_extended</code> and <code>zla_syrfxsx_extended</code> . Array, DIMENSION <i>nrhs</i> . <i>berr_out(j)</i> contains the componentwise relative backward error for right-hand-side <i>j</i> from the formula $\max(i) (\text{abs}(\text{res}(i)) / (\text{abs}(\text{op}(A)) * \text{abs}(y) + \text{abs}(B))(i))$ where $\text{abs}(z)$ is the componentwise absolute value of the matrix or vector <i>Z</i> . This is computed by <code>?la_lin_berr</code> .
<i>err_bnds_norm</i> , <i>err_bnds_comp</i>	Values of the corresponding input parameters improved after iterative refinement and stored in the second column of the array (1: <i>nrhs</i> , 2). The other elements are kept unchanged.
<i>info</i>	INTEGER. If <i>info</i> = 0, the execution is successful. The solution to every right-hand side is guaranteed. If <i>info</i> = - <i>i</i> , the <i>i</i> -th parameter had an illegal value.

See Also

[?syrfxsx](#)
[?sytrf](#)
[?sytrs](#)
[?lamch](#)
[ilaprec](#)
[ilatrans](#)
[?la_lin_berr](#)

[?la_syrpvgrw](#)

Computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$ for a symmetric indefinite matrix.

Syntax

FORTRAN 77:

```
call sla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
call dla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
call cla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
call zla_syrpvgrw( uplo, n, info, a, lda, af, ldaf, ipiv, work )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?la_syrpvgrw routine computes the reciprocal pivot growth factor $\text{norm}(A) / \text{norm}(U)$. The *max absolute element* norm is used. If this is much less than 1, the stability of the LU factorization of the equilibrated matrix A could be poor. This also means that the solution X , estimated condition numbers, and error bounds could be unreliable.

Input Parameters

<i>uplo</i>	CHARACTER*1. Must be 'U' or 'L'. Specifies the triangle of A to store: If <i>uplo</i> = 'U', the upper triangle of A is stored, If <i>uplo</i> = 'L', the lower triangle of A is stored.
<i>n</i>	INTEGER. The number of linear equations, the order of the matrix A ; $n \geq 0$.
<i>info</i>	INTEGER. The value of INFO returned from ?sytrf, that is, the pivot in column <i>info</i> is exactly 0.
<i>a</i> , <i>af</i>	REAL for sla_syrpvgrw DOUBLE PRECISION for dla_syrpvgrw COMPLEX for cla_syrpvgrw DOUBLE COMPLEX for zla_syrpvgrw. Arrays: <i>a</i> (<i>lda</i> , *), <i>af</i> (<i>ldaf</i> , *). The array <i>a</i> contains the input n -by- n matrix A . The second dimension of <i>a</i> must be at least $\max(1, n)$. The array <i>af</i> contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?sytrf. The second dimension of <i>af</i> must be at least $\max(1, n)$.
<i>lda</i>	INTEGER. The leading dimension of <i>a</i> ; $\text{lda} \geq \max(1, n)$.
<i>ldaf</i>	INTEGER. The leading dimension of <i>af</i> ; $\text{ldaf} \geq \max(1, n)$.
<i>ipiv</i>	INTEGER. Array, DIMENSION n . Details of the interchanges and the block structure of D as determined by ?sytrf.
<i>work</i>	REAL for sla_syrpvgrw and cla_syrpvgrw DOUBLE PRECISION for dla_syrpvgrw and zla_syrpvgrw. Workspace array, dimension $2*n$.

See Also

?sytrf

?la_wwaddw

Adds a vector into a doubled-single vector.

Syntax**FORTRAN 77:**

```
call sla_wwaddw( n, x, y, w )
call dla_wwaddw( n, x, y, w )
call cla_wwaddw( n, x, y, w )
call zla_wwaddw( n, x, y, w )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The ?la_wwaddw routine adds a vector W into a doubled-single vector (X, Y) . This works for all existing IBM hex and binary floating-point arithmetics, but not for decimal.

Input Parameters

<i>n</i>	INTEGER. The length of vectors X , Y , and W .
<i>x, y, w</i>	REAL for <code>sla_wwaddw</code> DOUBLE PRECISION for <code>dla_wwaddw</code> COMPLEX for <code>cla_wwaddw</code> DOUBLE COMPLEX for <code>zla_wwaddw</code> . Arrays DIMENSION <i>n</i> . <i>x</i> and <i>y</i> contain the first and second parts of the doubled-single accumulation vector, respectively. <i>w</i> contains the vector W to be added.

Output Parameters

<i>x, y</i>	Contain the first and second parts of the doubled-single accumulation vector, respectively, after adding the vector W .
-------------	---

Utility Functions and Routines

This section describes LAPACK utility functions and routines. Summary information about these routines is given in the following table:

LAPACK Utility Routines

Routine Name	Data Types	Description
ilaver		Returns the version of the Lapack library.

Routine Name	Data Types	Description
<code>ilaenv</code>		Environmental enquiry function which returns values for tuning algorithmic performance.
<code>iparmq</code>		Environmental enquiry function which returns values for tuning algorithmic performance.
<code>ieeeck</code>		Checks if the infinity and NaN arithmetic is safe. Called by <code>ilaenv</code> .
<code>?labad</code>	s, d	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
<code>?lamch</code>	s, d	Determines machine parameters for floating-point arithmetic.
<code>?lamc1</code>	s, d	Called from <code>?lamc2</code> . Determines machine parameters given by <i>beta</i> , <i>t</i> , <i>rnd</i> , <i>ieee1</i> .
<code>?lamc2</code>	s, d	Used by <code>?lamch</code> . Determines machine parameters specified in its arguments list.
<code>?lamc3</code>	s, d	Called from <code>?lamc1</code> - <code>?lamc5</code> . Intended to force <i>a</i> and <i>b</i> to be stored prior to doing the addition of <i>a</i> and <i>b</i> .
<code>?lamc4</code>	s, d	This is a service routine for <code>?lamc2</code> .
<code>?lamc5</code>	s, d	Called from <code>?lamc2</code> . Attempts to compute the largest machine floating-point number, without overflow.
<code>chla_transtype</code>		Translates a BLAST-specified integer constant to the character string specifying a transposition operation.
<code>iladiag</code>		Translates a character string specifying whether a matrix has a unit diagonal or not to the relevant BLAST-specified integer constant.
<code>ilaprec</code>		Translates a character string specifying an intermediate precision to the relevant BLAST-specified integer constant.
<code>ilatrans</code>		Translates a character string specifying a transposition operation to the BLAST-specified integer constant.
<code>ilauplo</code>		Translates a character string specifying an upper- or lower-triangular matrix to the relevant BLAST-specified integer constant.
<code>xerbla_array</code>		Assists other languages in calling the <code>xerbla</code> function.

See Also

`Isame` Tests two characters for equality regardless of the case.

`Isamen` Tests two character strings for equality regardless of the case.

`second/dsecnd` Returns elapsed time in seconds. Use to estimate real time between two calls to this function.

`xerbla` Error handling function called by BLAS, LAPACK, VML, and VSL functions.

ilaver

Returns the version of the LAPACK library.

Syntax

```
call ilaver( vers_major, vers_minor, vers_patch )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This routine returns the version of the LAPACK library.

Output Parameters

<code>vers_major</code>	INTEGER. Returns the major version of the LAPACK library.
<code>vers_minor</code>	INTEGER. Returns the minor version from the major version of the LAPACK library.
<code>vers_patch</code>	INTEGER. Returns the patch version from the minor version of the LAPACK library.

ilaenv

Environmental enquiry function that returns values for tuning algorithmic performance.

Syntax

```
value = ilaenv( ispec, name, opts, n1, n2, n3, n4 )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The enquiry function `ilaenv` is called from the LAPACK routines to choose problem-dependent parameters for the local environment. See `ispec` below for a description of the parameters.

This version provides a set of parameters that should give good, but not optimal, performance on many of the currently available computers.

This routine will not function correctly if it is converted to all lower case. Converting it to all upper case is allowed.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

ispec

INTEGER.

Specifies the parameter to be returned as the value of `ilaenv`:

- = 1: the optimal blocksize; if this value is 1, an unblocked algorithm will give the best performance.
- = 2: the minimum block size for which the block routine should be used; if the usable block size is less than this value, an unblocked routine should be used.
- = 3: the crossover point (in a block routine, for n less than this value, an unblocked routine should be used)
- = 4: the number of shifts, used in the nonsymmetric eigenvalue routines (deprecated)
- = 5: the minimum column dimension for blocking to be used; rectangular blocks must have dimension at least k -by- m , where k is given by `ilaenv(2, ...)` and m by `ilaenv(5, ...)`
- = 6: the crossover point for the SVD (when reducing an m -by- n matrix to bidiagonal form, if $\max(m, n)/\min(m, n)$ exceeds this value, a QR factorization is used first to reduce the matrix to a triangular form.)
- = 7: the number of processors
- = 8: the crossover point for the multishift QR and QZ methods for nonsymmetric eigenvalue problems (deprecated).
- = 9: maximum size of the subproblems at the bottom of the computation tree in the divide-and-conquer algorithm (used by `?gelsd` and `?gesdd`)
- =10: ieee NaN arithmetic can be trusted not to trap
- =11: infinity arithmetic can be trusted not to trap
- $12 \leq ispec \leq 16$: `?hseqr` or one of its subroutines, see `iparmq` for detailed explanation.

name

CHARACTER* (*). The name of the calling subroutine, in either upper case or lower case.

opts

CHARACTER* (*). The character options to the subroutine *name*, concatenated into a single character string. For example, *uplo* = 'U', *trans* = 'T', and *diag* = 'N' for a triangular routine would be specified as *opts* = 'UTN'.

n1, n2, n3, n4

INTEGER. Problem dimensions for the subroutine *name*; these may not all be required.

Output Parameters

value

INTEGER.

If *value* ≥ 0 : the value of the parameter specified by *ispec*;

If *value* = $-k < 0$: the k -th argument had an illegal value.

Application Notes

The following conventions have been used when calling `ilaenv` from the LAPACK routines:

1. `opts` is a concatenation of all of the character options to subroutine `name`, in the same order that they appear in the argument list for `name`, even if they are not used in determining the value of the parameter specified by `ispec`.
2. The problem dimensions n_1, n_2, n_3, n_4 are specified in the order that they appear in the argument list for `name`. n_1 is used first, n_2 second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by `ilaenv` is checked for validity in the calling subroutine. For example, `ilaenv` is used to retrieve the optimal blocksize for `strtri` as follows:

```
nb = ilaenv( 1, 'strtri', uplo // diag, n, -1, -1, -1> )
if( nb.le.1 ) nb = max( 1, n )
```

Below is an example of `ilaenv` usage in C language:

```
#include <stdio.h>
#include "mkl.h"

int main(void)
{
    int size = 1000;
    int ispec = 1;
    int dummy = -1;

    int blockSize1 = ilaenv(&ispec, "dsytrd", "U", &size, &dummy, &dummy);
    int blockSize2 = ilaenv(&ispec, "dormtr", "LUN", &size, &size, &dummy, &dummy);

    printf("DSYTRD blocksize = %d\n",
blockSize1);
    printf("DORMTR blocksize = %d\n", blockSize2);

    return 0;
}
```

See Also

[?hseqr](#)
[iparmq](#)

iparmq

Environmental enquiry function which returns values for tuning algorithmic performance.

Syntax

`value = iparmq(ispec, name, opts, n, ilo, ihi, lwork)`

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The function sets problem and machine dependent parameters useful for `?hseqr` and its subroutines. It is called whenever `ilaenv` is called with $12 \leq ispec \leq 16$.

Input Parameters

<i>ispec</i>	INTEGER. Specifies the parameter to be returned as the value of <code>iparmq</code> : = 12: (<i>inmin</i>) Matrices of order <i>nmin</i> or less are sent directly to <code>?lahqr</code> , the implicit double shift QR algorithm. <i>nmin</i> must be at least 11. = 13: (<i>inwin</i>) Size of the deflation window. This is best set greater than or equal to the number of simultaneous shifts <i>ns</i> . Larger matrices benefit from larger deflation windows. = 14: (<i>inibl</i>) Determines when to stop nibbling and invest in an (expensive) multi-shift QR sweep. If the aggressive early deflation subroutine finds <i>ld</i> converged eigenvalues from an order <i>nw</i> deflation window and $ld > (nw * nibble) / 100$, then the next QR sweep is skipped and early deflation is applied immediately to the remaining active diagonal block. Setting <code>iparmq(ispec=14)=0</code> causes <code>TTQRE</code> to skip a multi-shift QR sweep whenever early deflation finds a converged eigenvalue. Setting <code>iparmq(ispec=14)</code> greater than or equal to 100 prevents <code>TTQRE</code> from skipping a multi-shift QR sweep. = 15: (<i>nshfts</i>) The number of simultaneous shifts in a multi-shift QR iteration. = 16: (<i>iacc22</i>) <code>iparmq</code> is set to 0, 1 or 2 with the following meanings. 0: During the multi-shift QR sweep, <code>?laqr5</code> does not accumulate reflections and does not use matrix-matrix multiply to update the far-from-diagonal matrix entries. 1: During the multi-shift QR sweep, <code>?laqr5</code> and/or <code>?laqr3</code> accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries. 2: During the multi-shift QR sweep, <code>?laqr5</code> accumulates reflections and takes advantage of 2-by-2 block structure during matrix-matrix multiplies. (If <code>?trrm</code> is slower than <code>?gemm</code> , then <code>iparmq(ispec=16)=1</code> may be more efficient than <code>iparmq(ispec=16)=2</code> despite the greater level of arithmetic work implied by the latter choice.)
<i>name</i>	CHARACTER* (*). The name of the calling subroutine.
<i>opts</i>	CHARACTER* (*). This is a concatenation of the string arguments to <code>TTQRE</code> .
<i>n</i>	INTEGER. <i>n</i> is the order of the Hessenberg matrix <i>H</i> .
<i>ilo, ihi</i>	INTEGER. It is assumed that <i>H</i> is already upper triangular in rows and columns 1: <i>ilo-1</i> and <i>ihi+1:n</i> .
<i>lwork</i>	INTEGER.

The amount of workspace available.

Output Parameters

<i>value</i>	INTEGER. If <i>value</i> ≥ 0 : the value of the parameter specified by <i>iparmq</i> ; If <i>value</i> = $-k < 0$: the <i>k</i> -th argument had an illegal value.
--------------	--

Application Notes

The following conventions have been used when calling `ilaenv` from the LAPACK routines:

1. *opts* is a concatenation of all of the character options to subroutine *name*, in the same order that they appear in the argument list for *name*, even if they are not used in determining the value of the parameter specified by *ispec*.
2. The problem dimensions *n1*, *n2*, *n3*, *n4* are specified in the order that they appear in the argument list for *name*. *n1* is used first, *n2* second, and so on, and unused problem dimensions are passed a value of -1.
3. The parameter value returned by `ilaenv` is checked for validity in the calling subroutine. For example, `ilaenv` is used to retrieve the optimal blocksize for `strtri` as follows:

```
nb = ilaenv( 1, 'strtri', uplo // diag, n, -1, -1, -1> )
if( nb.le.1 ) nb = max( 1, n )
```

ieeeck

Checks if the infinity and NaN arithmetic is safe.

Called by `ilaenv`.

Syntax

```
ival = ieeeck( ispec, zero, one )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The function `ieeeck` is called from `ilaenv` to verify that infinity and possibly NaN arithmetic is safe, that is, will not trap.

Input Parameters

<i>ispec</i>	INTEGER. Specifies whether to test just for infinity arithmetic or both for infinity and NaN arithmetic: If <i>ispec</i> = 0: Verify infinity arithmetic only. If <i>ispec</i> = 1: Verify infinity and NaN arithmetic.
<i>zero</i>	REAL. Must contain the value 0.0 This is passed to prevent the compiler from optimizing away this code.

<i>one</i>	REAL. Must contain the value 1.0 This is passed to prevent the compiler from optimizing away this code.
------------	--

Output Parameters

<i>ival</i>	INTEGER. If <i>ival</i> = 0: Arithmetic failed to produce the correct answers. If <i>ival</i> = 1: Arithmetic produced the correct answers.
-------------	---

?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

```
call slabad( small, large )
call dlabad( small, large )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine takes as input the values computed by `slamch/dlamch` for underflow and overflow, and returns the square root of each of these values if the log of *large* is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by `?lamch`. This subroutine is needed because `?lamch` does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

Input Parameters

<i>small</i>	REAL for slabad DOUBLE PRECISION for dlabad. The underflow threshold as computed by <code>?lamch</code> .
<i>large</i>	REAL for slabad DOUBLE PRECISION for dlabad. The overflow threshold as computed by <code>?lamch</code> .

Output Parameters

<i>small</i>	On exit, if $\log_{10}(\text{large})$ is sufficiently large, the square root of <i>small</i> , otherwise unchanged.
<i>large</i>	On exit, if $\log_{10}(\text{large})$ is sufficiently large, the square root of <i>large</i> , otherwise unchanged.

?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

```
val = slamch( cmach )
```

```
val = dlamch( cmach )
```

C:

```
<datatype> LAPACKE_<?>lamch (char cmach);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The function ?lamch determines single precision and double precision machine parameters.

Input Parameters

<i>cmach</i>	CHARACTER*1. Specifies the value to be returned by ?lamch: = 'E' or 'e', <i>val</i> = <i>eps</i> = 'S' or 's', <i>val</i> = <i>smin</i> = 'B' or 'b', <i>val</i> = <i>base</i> = 'P' or 'p', <i>val</i> = <i>eps</i> * <i>base</i> = 'n' or 'n', <i>val</i> = <i>t</i> = 'R' or 'r', <i>val</i> = <i>rnd</i> = 'M' or 'm', <i>val</i> = <i>emin</i> = 'U' or 'u', <i>val</i> = <i>rmin</i> = 'L' or 'l', <i>val</i> = <i>emax</i> = 'O' or 'o', <i>val</i> = <i>rmax</i> where <i>eps</i> = relative machine precision; <i>smin</i> = safe minimum, such that $1/smin$ does not overflow; <i>base</i> = base of the machine; <i>prec</i> = <i>eps</i> * <i>base</i> ; <i>t</i> = number of (<i>base</i>) digits in the mantissa; <i>rnd</i> = 1.0 when rounding occurs in addition, 0.0 otherwise; <i>emin</i> = minimum exponent before (gradual) underflow; <i>rmin</i> = <i>underflow_threshold</i> - <i>base</i> ^(<i>emin</i>-1) ; <i>emax</i> = largest exponent before overflow; <i>rmax</i> = <i>overflow_threshold</i> - (<i>base</i> ^{<i>emax</i>})*(1- <i>eps</i>).
--------------	---

NOTE

You can use a character string for *cmach* instead of a single character in order to make your code more readable. The first character of the string determines the value to be returned. For example, 'Precision' is interpreted as 'p'.

Output Parameters

<i>val</i>	REAL for <code>slamch</code>
	DOUBLE PRECISION for <code>dlamch</code>
	Value returned by the function.

?lamc1

Called from ?lamc2. Determines machine parameters given by beta, t, rnd, ieee1.

Syntax

```
call slamc1( beta, t, rnd, ieee1 )
call dlamc1( beta, t, rnd, ieee1 )
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

The routine `?lamc1` determines machine parameters given by *beta, t, rnd, ieee1*.

Output Parameters

<i>beta</i>	INTEGER. The base of the machine.
<i>t</i>	INTEGER. The number of (<i>beta</i>) digits in the mantissa.
<i>rnd</i>	LOGICAL. Specifies whether proper rounding (<i>rnd</i> = .TRUE.) or chopping (<i>rnd</i> = .FALSE.) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.
<i>ieee1</i>	LOGICAL. Specifies whether rounding appears to be done in the <i>ieee</i> 'round to nearest' style.

?lamc2

Used by ?lamch. Determines machine parameters specified in its arguments list.

Syntax

```
call slamc2( beta, t, rnd, eps, emin, rmin, emax, rmax )
call dlamc2( beta, t, rnd, eps, emin, rmin, emax, rmax )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lamc2 determines machine parameters specified in its arguments list.

Output Parameters

<i>beta</i>	INTEGER. The base of the machine.
<i>t</i>	INTEGER. The number of (<i>beta</i>) digits in the mantissa.
<i>rnd</i>	LOGICAL. Specifies whether proper rounding (<i>rnd</i> = .TRUE.) or chopping (<i>rnd</i> = .FALSE.) occurs in addition. This may not be a reliable guide to the way in which the machine performs its arithmetic.
<i>eps</i>	REAL for slamc2 DOUBLE PRECISION for dlamc2 The smallest positive number such that $f1(1.0 - \text{eps}) < 1.0$, where <i>f1</i> denotes the computed value.
<i>emin</i>	INTEGER. The minimum exponent before (gradual) underflow occurs.
<i>rmin</i>	REAL for slamc2 DOUBLE PRECISION for dlamc2 The smallest normalized number for the machine, given by $\text{base}^{\text{emin}-1}$, where <i>base</i> is the floating point value of <i>beta</i> .
<i>emax</i>	INTEGER. The maximum exponent before overflow occurs.
<i>rmax</i>	REAL for slamc2 DOUBLE PRECISION for dlamc2 The largest positive number for the machine, given by $\text{base}^{\text{emax}(1 - \text{eps})}$, where <i>base</i> is the floating point value of <i>beta</i> .

?lamc3

Called from ?lamc1-?lamc5. Intended to force *a* and *b* to be stored prior to doing the addition of *a* and *b*.

Syntax

```
val = slamc3( a, b )
val = dlamc3( a, b )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine is intended to force A and B to be stored prior to doing the addition of A and B , for use in situations where optimizers might hold one of these in a register.

Input Parameters

a, b	REAL for slamc3 DOUBLE PRECISION for dlamc3
	The values a and b .

Output Parameters

val	REAL for slamc3 DOUBLE PRECISION for dlamc3
	The result of adding values a and b .

?lamc4

This is a service routine for ?lamc2.

Syntax

```
call slamc4( emin, start, base )
call dlamc4( emin, start, base )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

This is a service routine for ?lamc2.

Input Parameters

$start$	REAL for slamc4 DOUBLE PRECISION for dlamc4
	The starting point for determining $emin$.
$base$	INTEGER. The base of the machine.

Output Parameters

emin INTEGER. The minimum exponent before (gradual) underflow, computed by setting $a = start$ and dividing by $base$ until the previous a can not be recovered.

?lamc5

Called from ?lamc2. Attempts to compute the largest machine floating-point number, without overflow.

Syntax

```
call slamc5( beta, p, emin, ieee, emax, rmax)
call dlamc5( beta, p, emin, ieee, emax, rmax)
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine ?lamc5 attempts to compute $rmax$, the largest machine floating-point number, without overflow. It assumes that $emax + \text{abs}(emin)$ sum approximately to a power of 2. It will fail on machines where this assumption does not hold, for example, the Cyber 205 ($emin = -28625$, $emax = 28718$). It will also fail if the value supplied for $emin$ is too large (that is, too close to zero), probably with overflow.

Input Parameters

<i>beta</i>	INTEGER. The base of floating-point arithmetic.
<i>p</i>	INTEGER. The number of base <i>beta</i> digits in the mantissa of a floating-point value.
<i>emin</i>	INTEGER. The minimum exponent before (gradual) underflow.
<i>ieee</i>	LOGICAL. A logical flag specifying whether or not the arithmetic system is thought to comply with the IEEE standard.

Output Parameters

<i>emax</i>	INTEGER. The largest exponent before overflow.
<i>rmax</i>	REAL for slamc5 DOUBLE PRECISION for dlamc5 The largest machine floating-point number.

chla_transtype

Translates a BLAST-specified integer constant to the character string specifying a transposition operation.

Syntax

```
val = chla_transtype( trans )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `chla_transtype` function translates a BLAST-specified integer constant to the character string specifying a transposition operation.

The function returns a `CHARACTER*1`. If the input is not an integer indicating a transposition operator, then `val` is '`x`'. Otherwise, the function returns the constant value corresponding to `trans`.

Input Parameters

<code>trans</code>	INTEGER. Specifies the form of the system of equations: If <code>trans</code> = <code>BLAS_NO_TRANS</code> = 111: No transpose. If <code>trans</code> = <code>BLAS_TRANS</code> = 112: Transpose. If <code>trans</code> = <code>BLAS_CONJ_TRANS</code> = 113: Conjugate Transpose.
--------------------	--

Output Parameters

<code>val</code>	<code>CHARACTER*1</code> Character that specifies a transposition operation.
------------------	---

iladiag

Translates a character string specifying whether a matrix has a unit diagonal to the relevant BLAST-specified integer constant.

Syntax

```
val = iladiag( diag )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `iladiag` function translates a character string specifying whether a matrix has a unit diagonal or not to the relevant BLAST-specified integer constant.

The function returns an `INTEGER`. If `val` < 0, the input is not a character indicating a unit or non-unit diagonal. Otherwise, the function returns the constant value corresponding to `diag`.

Input Parameters

<code>diag</code>	<code>CHARACTER*1</code> .
-------------------	----------------------------

Specifies the form of the system of equations:

If $diag = 'N'$: A is non-unit triangular.

If $diag = 'U'$: A is unit triangular.

Output Parameters

`val` INTEGER

Value returned by the function.

ilaprec

Translates a character string specifying an intermediate precision to the relevant BLAST-specified integer constant.

Syntax

`val = ilaprec(prec)`

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `ilaprec` function translates a character string specifying an intermediate precision to the relevant BLAST-specified integer constant.

The function returns an INTEGER. If $val < 0$, the input is not a character indicating a supported intermediate precision. Otherwise, the function returns the constant value corresponding to `prec`.

Input Parameters

`prec` CHARACTER*1.

Specifies the form of the system of equations:

If $prec = 'S'$: Single.

If $prec = 'D'$: Double.

If $prec = 'I'$: Indigenous.

If $prec = 'X', 'E'$: Extra.

Output Parameters

`val` INTEGER

Value returned by the function.

ilatrans

Translates a character string specifying a transposition operation to the BLAST-specified integer constant.

Syntax

```
val = ilatrans( trans )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `ilatrans` function translates a character string specifying a transposition operation to the BLAST-specified integer constant.

The function returns a `INTEGER`. If `val < 0`, the input is not a character indicating a transposition operator. Otherwise, the function returns the constant value corresponding to `trans`.

Input Parameters

<code>trans</code>	<code>CHARACTER*1.</code>
Specifies the form of the system of equations:	
If <code>trans = 'N'</code> :	No transpose.
If <code>trans = 'T'</code> :	Transpose.
If <code>trans = 'C'</code> :	Conjugate Transpose.

Output Parameters

<code>val</code>	<code>INTEGER</code>
Character that specifies a transposition operation.	

ilauplo

Translates a character string specifying an upper- or lower-triangular matrix to the relevant BLAST-specified integer constant.

Syntax

```
val = ilauplo( uplo )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The `ilauplo` function translates a character string specifying an upper- or lower-triangular matrix to the relevant BLAST-specified integer constant.

The function returns an `INTEGER`. If `val < 0`, the input is not a character indicating an upper- or lower-triangular matrix. Otherwise, the function returns the constant value corresponding to `uplo`.

Input Parameters

<code>diag</code>	<code>CHARACTER.</code>
-------------------	-------------------------

Specifies the form of the system of equations:

If $\text{diag} = \text{'U'}$: A is upper triangular.

If $\text{diag} = \text{'L'}$: A is lower triangular.

Output Parameters

val INTEGER

Value returned by the function.

xerbla_array

Assists other languages in calling the xerbla function.

Syntax

```
call xerbla_array( srname_array, srname_len, info )
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routine assists other languages in calling the error handling `xerbla` function. Rather than taking a Fortran string argument as the function name, `xerbla_array` takes an array of single characters along with the array length. The routine then copies up to 32 characters of that array into a Fortran string and passes that to `xerbla`. If called with a non-positive `srname_len`, the routine will call `xerbla` with a string of all blank characters.

If some macro or other device makes `xerbla_array` available to C99 by a name `lapack_xerbla` and with a common Fortran calling convention, a C99 program could invoke `xerbla` via:

```
{
    int flen = strlen(__func__);
    lapack_xerbla(__func__, &flen, &info);
}
```

Providing `xerbla_array` is not necessary for intercepting LAPACK errors. `xerbla_array` calls `xerbla`.

Output Parameters

srname_array CHARACTER(1).

Array, dimension (*srname_len*). The name of the routine that called `xerbla_array`.

srname_len INTEGER.

The length of the name in `srname_array`.

info INTEGER.

Position of the invalid parameter in the parameter list of the calling routine.

Test Functions and Routines

This section describes LAPACK test functions and routines. Summary information about these routines is given in the following table:

LAPACK Test Routines

Routine Name	Data Types	Description
?lagge	s, d, c, z	Generates a general m -by- n matrix .
?laghe	c, z	Generates a complex Hermitian matrix .
?lagsy	s, d, c, z	Generates a symmetric matrix by pre- and post- multiplying a real diagonal matrix with a random unitary matrix .
?latms	s, d, c, z	Generates a general m -by- n matrix with specific singular values.

?lagge

Generates a general m -by- n matrix .

Syntax

C:

```
lapack_int LAPACKE_<?>lagge (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const <datatype> * d, <datatype> * a, lapack_int lda, lapack_int * iseed);
```

Include Files

- C: mkl.h

Description

The routine generates a general m -by- n matrix A , by pre- and post- multiplying a real diagonal matrix D with random matrices U and V :

$$A := U^* D^* V,$$

where U and V are orthogonal for real flavors and unitary for complex flavors. The lower and upper bandwidths may then be reduced to kl and ku by additional orthogonal transformations.

Input Parameters

The data types are given for the Fortran interface.

<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns of the matrix A ($n \geq 0$).
<i>kl</i>	INTEGER. The number of nonzero subdiagonals within the band of A ($0 \leq kl \leq m-1$).
<i>ku</i>	INTEGER. The number of nonzero superdiagonals within the band of A ($0 \leq ku \leq n-1$).
<i>d</i>	REAL for slagge, clagge

DOUBLE PRECISION for dlagge, zlagge.

The array d with the dimension of ($\min(m, n)$) contains the diagonal elements of the diagonal matrix D .

lda INTEGER. The leading dimension of the array a ($lda \geq m$).

$iseed$ INTEGER.

The array $iseed$ with the dimension of 4 contains the seed of the random number generator. The elements must be between 0 and 4095 and $iseed(4)$ must be odd.

Output Parameters

a REAL for slagge

DOUBLE PRECISION for dlagge

COMPLEX for clagge

DOUBLE COMPLEX for zlagge.

The array a with dimensions of (lda, n) contains the generated m -by- n matrix D .

$iseed$ The array $iseed$ contains the updated seed on exit.

?laghe

Generates a complex Hermitian matrix .

Syntax

C:

```
lapack_int LAPACKE_<?>laghe (int matrix_layout, lapack_int n, lapack_int k, const
<datatype> * d, <datatype> * a, lapack_int lda, lapack_int * iseed);
```

Include Files

- C: mkl.h

Description

The routine generates a complex Hermitian matrix A , by pre- and post- multiplying a complex diagonal matrix D with random unitary matrix:

$$A := U^* D^* U^T$$

The semi-bandwidth may then be reduced to k by additional unitary transformations.

Input Parameters

The data types are given for the Fortran interface.

n INTEGER. The order of the matrix A ($n \geq 0$).

k INTEGER. The number of nonzero subdiagonals within the band of A ($0 \leq k \leq n-1$).

<i>d</i>	COMPLEX for claghe DOUBLE COMPLEX for zlaghe.
	The array <i>d</i> with the dimension of (<i>n</i>) contains the diagonal elements of the diagonal matrix <i>D</i> .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> (<i>lda</i> $\geq n$).
<i>iseed</i>	INTEGER. The array <i>iseed</i> with the dimension of 4 contains the seed of the random number generator. The elements must be between 0 and 4095 and <i>iseed</i> (4) must be odd.

Output Parameters

<i>a</i>	COMPLEX for claghe DOUBLE COMPLEX for zlaghe.
	The array <i>a</i> with dimensions of (<i>lda</i> , <i>n</i>) contains the generated <i>n</i> -by- <i>n</i> Hermitian matrix <i>D</i> .
<i>iseed</i>	The array <i>iseed</i> contains the updated seed on exit.

?lagsy

Generates a symmetric matrix by pre- and post-multiplying a real diagonal matrix with a random unitary matrix .

Syntax

C:

```
lapack_int LAPACKE_<?>lagsy (int matrix_layout, lapack_int n, lapack_int k, const
<datatype> * d, <datatype> * a, lapack_int lda, lapack_int * iseed);
```

Include Files

- C: mkl.h

Description

SLAGSY generates a symmetric matrix *A* by pre- and post- multiplying a real diagonal matrix *D* with a random matrix *U*:

$$A := U^* D^* U^T,$$

where *U* is orthogonal for real flavors and unitary for complex flavors. The semi-bandwidth may then be reduced to *k* by additional unitary transformations.

Input Parameters

The data types are given for the Fortran interface.

<i>n</i>	INTEGER. The order of the matrix <i>A</i> (<i>n</i> ≥ 0).
----------	--

<i>k</i>	INTEGER. The number of nonzero subdiagonals within the band of A ($0 \leq k \leq n-1$).
<i>d</i>	REAL for slagsy, clagsy DOUBLE PRECISION for dlagsy, zlagsy. The array <i>d</i> with the dimension of (<i>n</i>) contains the diagonal elements of the diagonal matrix D .
<i>lda</i>	INTEGER. The leading dimension of the array <i>a</i> (<i>lda</i> $\geq n$).
<i>iseed</i>	INTEGER. The array <i>iseed</i> with the dimension of 4 contains the seed of the random number generator. The elements must be between 0 and 4095 and <i>iseed</i> (4) must be odd.

Output Parameters

<i>a</i>	REAL for slagsy DOUBLE PRECISION for dlagsy COMPLEX for clagsy DOUBLE COMPLEX for zlagsy. The array <i>a</i> with dimensions of (<i>lda</i> , <i>n</i>) contains the generated symmetric <i>n</i> -by- <i>n</i> matrix D .
<i>iseed</i>	The array <i>iseed</i> contains the updated seed on exit.

?latms

Generates a general *m*-by-*n* matrix with specific singular values.

Syntax

Fortran:

```
call slatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)
call dlatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)
call clatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)
call zlatms (m, n, dist, iseed, sym, d, mode, cond, dmax, kl, ku, pack, a, lda, work, info)
```

C:

```
lapack_int LAPACKE_slatms (int matrix_layout, lapack_int m, lapack_int n, char dist,
lapack_int * iseed, char sym, float * d, lapack_int mode, float cond, float dmax,
lapack_int kl, lapack_int ku, char pack, float * a, lapack_int lda);
```

```
lapack_int LAPACKE_dlatms (int matrix_layout, lapack_int m, lapack_int n, char dist,
lapack_int * iseed, char sym, double * d, lapack_int mode, double cond, double dmax,
lapack_int kl, lapack_int ku, char pack, double * a, lapack_int lda);

lapack_int LAPACKE_clatms (int matrix_layout, lapack_int m, lapack_int n, char dist,
lapack_int * iseed, char sym, float * d, lapack_int mode, float cond, float dmax,
lapack_int kl, lapack_int ku, char pack, lapack_complex_float * a, lapack_int lda);

lapack_int LAPACKE_zlatms (int matrix_layout, lapack_int m, lapack_int n, char dist,
lapack_int * iseed, char sym, double * d, lapack_int mode, double cond, double dmax,
lapack_int kl, lapack_int ku, char pack, lapack_complex_double * a, lapack_int lda);
```

Include Files

- C: mkl.h

Description

The ?latms routine generates random matrices with specified singular values, or symmetric/Hermitian matrices with specified eigenvalues for testing LAPACK programs.

It applies this sequence of operations:

1. Set the diagonal to d , where d is input or computed according to $mode$, $cond$, $dmax$, and sym as described in Input Parameters.
2. Generate a matrix with the appropriate band structure, by one of two methods:

Method A

1. Generate a dense m -by- n matrix by multiplying d on the left and the right by random unitary matrices, then:
2. Reduce the bandwidth according to kl and ku , using Householder transformations.

Method B:

Convert the bandwidth-0 (i.e., diagonal) matrix to a bandwidth-1 matrix using Givens rotations, "chasing" out-of-band elements back, much as in QR; then convert the bandwidth-1 to a bandwidth-2 matrix, etc.

Note that for reasonably small bandwidths (relative to m and n) this requires less storage, as a dense matrix is not generated. Also, for symmetric or Hermitian matrices, only one triangle is generated.

Method A is chosen if the bandwidth is a large fraction of the order of the matrix, and lda is at least m (so a dense matrix can be stored.) Method B is chosen if the bandwidth is small (less than $(1/2)*n$ for symmetric or Hermitian or less than $.3*n+m$ for nonsymmetric), or lda is less than m and not less than the bandwidth.

Pack the matrix if desired, using one of the methods specified by the $pack$ parameter.

If Method B is chosen and band format is specified, then the matrix is generated in the band format and no repacking is necessary.

Input Parameters

The data types are given for the Fortran interface.

m

INTEGER. The number of rows of the matrix A ($m \geq 0$).

n

INTEGER. The number of columns of the matrix A ($n \geq 0$). If the matrix is not symmetric ($sym \neq 'N'$) n must equal m .

<i>dist</i>	CHARACTER*1. Specifies the type of distribution to be used to generate the random singular values or eigenvalues:
	<ul style="list-style-type: none"> • 'U': uniform distribution (0, 1) • 'S': symmetric uniform distribution (-1, 1) • 'N': normal distribution (0, 1)
<i>iseed</i>	INTEGER. Array with size 4.
	Specifies the seed of the random number generator. Values should lie between 0 and 4095 inclusive, and <i>iseed</i> (4) should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers. The values of the array are modified, and can be used in the next call to ?latms to continue the same random number sequence.
<i>sym</i>	CHARACTER*1.
	If <i>sym</i> ='S' or 'H', the generated matrix is symmetric or Hermitian, with eigenvalues specified by <i>d</i> , <i>cond</i> , <i>mode</i> , and <i>dmax</i> ; they can be positive, negative, or zero.
	If <i>sym</i> ='P', the generated matrix is symmetric or Hermitian, with eigenvalues (which are singular, non-negative values) specified by <i>d</i> , <i>cond</i> , <i>mode</i> , and <i>dmax</i> .
	If <i>sym</i> ='N', the generated matrix is nonsymmetric, with singular, non-negative values specified by <i>d</i> , <i>cond</i> , <i>mode</i> , and <i>dmax</i> .
<i>d</i>	REAL for slatms and clatms DOUBLE PRECISION for dlatms and zlatms Array, size (MIN(<i>m</i> , <i>n</i>))
	This array is used to specify the singular values or eigenvalues of A (see the description of <i>sym</i>). If <i>mode</i> =0, then <i>d</i> is assumed to contain the eigenvalues or singular values, otherwise elements of <i>d</i> are computed according to <i>mode</i> , <i>cond</i> , and <i>dmax</i> .
<i>mode</i>	INTEGER. Describes how the singular/eigenvalues are specified.
	<ul style="list-style-type: none"> • <i>mode</i> = 0: use <i>d</i> as input • <i>mode</i> = 1: set <i>d</i>(1) = 1 and <i>d</i>(2:<i>n</i>) = 1.0/<i>cond</i> • <i>mode</i> = 2: set <i>d</i>(1:<i>n</i>-1) = 1 and <i>d</i>(<i>n</i>) = 1.0/<i>cond</i> • <i>mode</i> = 3: set <i>d</i>(<i>i</i>) = <i>cond</i>^{-(<i>i</i>-1)/(<i>n</i>-1)} • <i>mode</i> = 4: set <i>d</i>(<i>i</i>) = 1 - (<i>i</i>-1)/(<i>n</i>-1)*(1-1/<i>cond</i>) • <i>mode</i> = 5: set elements of <i>d</i> to random numbers in the range (1/<i>cond</i>, 1) such that their logarithms are uniformly distributed. • <i>mode</i> = 6: set elements of <i>d</i> to random numbers from same distribution as the rest of the matrix.
	<i>mode</i> < 0 has the same meaning as ABS(<i>mode</i>), except that the order of the elements of <i>d</i> is reversed. Thus, if <i>mode</i> is positive, <i>d</i> has entries ranging from 1 to 1/ <i>cond</i> , if negative, from 1/ <i>cond</i> to 1.
	If <i>sym</i> ='S' or 'H', and <i>mode</i> is not 0, 6, nor -6, then the elements of <i>d</i> are also given a random sign (multiplied by +1 or -1).

cond REAL for `slatms` and `clatms`
 DOUBLE PRECISION for `dlatms` and `zlatms`
 Used in setting *d* as described for the *mode* parameter. If used, *cond* ≥ 1 .

dmax REAL for `slatms` and `clatms`
 DOUBLE PRECISION for `dlatms` and `zlatms`
 If *mode* is not -6, 0 nor 6, the contents of *d*, as computed according to *mode* and *cond*, are scaled by *dmax* / $\max(\text{abs}(d(i)))$; thus, the maximum absolute eigenvalue or singular value (the norm) is $\text{abs}(dmax)$.

NOTE

dmax need not be positive: if *dmax* is negative (or zero), *d* will be scaled by a negative number (or zero).

kl INTEGER. Specifies the lower bandwidth of the matrix. For example, *kl*=0 implies upper triangular, *kl*=1 implies upper Hessenberg, and *kl* being at least *m* - 1 means that the matrix has full lower bandwidth. *kl* must equal *ku* if the matrix is symmetric or Hermitian.

ku INTEGER. Specifies the upper bandwidth of the matrix. For example, *ku*=0 implies lower triangular, *ku*=1 implies lower Hessenberg, and *ku* being at least *n* - 1 means that the matrix has full upper bandwidth. *kl* must equal *ku* if the matrix is symmetric or Hermitian.

pack CHARACTER*1. Specifies packing of matrix:

- 'N': no packing
- 'U': zero out all subdiagonal entries (if symmetric or Hermitian)
- 'L': zero out all superdiagonal entries (if symmetric or Hermitian)
- 'B': store the lower triangle in band storage scheme (only if matrix symmetric, Hermitian, or lower triangular)
- 'Q': store the upper triangle in band storage scheme (only if matrix symmetric, Hermitian, or upper triangular)
- 'Z': store the entire matrix in band storage scheme (pivoting can be provided for by using this option to store *A* in the trailing rows of the allocated storage)

Using these options, the various LAPACK packed and banded storage schemes can be obtained:

	'Z'	'B'	'Q'	'C'	'R'
GB: general band			x		
PB: symmetric positive definite band		x	x		
SB: symmetric band	x		x		
HB: Hermitian band		x	x		
TB: triangular band	x		x		
PP: symmetric positive definite packed				x	x

If two calls to `?latms` differ only in the *pack* parameter, they generate mathematically equivalent matrices.

1da

INTEGER. *lda* specifies the first dimension of *a* as declared in the calling program.

If `pack='N', 'U', 'L', 'C', or 'R'`, then `lda` must be at least m .

If `pack='B'` or `'Q'`, then `lda` must be at least $\text{MIN}(k_1, m - 1)$ (which is equal to $\text{MIN}(k_u, n - 1)$).

If $\text{pack} = \text{'Z}'$, lda must be large enough to hold the packed array: $\text{MIN}(ku, n - 1) + \text{MIN}(kl, m - 1) + 1$.

Output Parameters

*i*seed

The array *iseed* contains the updated seed.

d

The array d contains the updated seed.

NOTE

The array d is not modified if $mode = 0$.

a

REAL for slatms

DOUBLE PRECISION for dlatms

COMPLEX for clatms

DOUBLE COMPLEX for zlatms

Array of size lda by n .

The array a contains the

a is first generated in full (unpacked) form, and then

by *pack*. Thus, the first *m* elements of the first *n* columns are always modified. If *pack* specifies a packed or banded storage scheme, all *lda*

elements of the first n columns are modified; the elements of the array which do not correspond to elements of the generated matrix are set to zero.

work

```
REAL for slatms
DOUBLE PRECISION for dlatms
COMPLEX for clatms
DOUBLE COMPLEX for zlatms
Array of size (3*MAX( $n, m$ ))
```

Workspace.

info

INTEGER. Error code. *info* is set to one of the following values:

- 0: normal return
- -1: m negative or unequal to n and $\text{sym}='S'$, ' H ', or ' P '
- -2: n negative
- -3: dist illegal string
- -5: sym illegal string
- -7: mode not in range -6 to 6
- -8: cond less than 1.0, and mode not -6, 0 nor 6
- -10: k_1 negative
- -11: k_u negative, or $\text{sym} \neq 'N'$ and k_u not equal to k_1
- -12: pack illegal string, or $\text{pack}='U'$ or ' L ', and $\text{sym}='N'$; or $\text{pack}='C'$ or ' Q ' and $\text{sym}='N'$ and k_1 is not zero; or $\text{pack}='R'$ or ' B ' and $\text{sym}='N'$ and k_u is not zero; or $\text{pack}='U', 'L', 'C', 'R', 'B'$, or ' Q ', and m is not n .
- 14: l_da is less than m , or $\text{pack}='Z'$ and l_da is less than $\text{MIN}(k_u, n - 1) + \text{MIN}(k_1, m - 1) + 1$.
- 2: Cannot scale to d_{\max} (maximum singular value is 0)
- 3: Error return from [lagge](#), [?laghe](#), or [lagsy](#)

ScaLAPACK Routines

This chapter describes the Intel® Math Kernel Library implementation of routines from the ScaLAPACK package for distributed-memory architectures. Routines are supported for both real and complex dense and band matrices to perform the tasks of solving systems of linear equations, solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

Intel MKL ScaLAPACK routines are written in FORTRAN 77 with exception of a few utility routines written in C to exploit the IEEE arithmetic. All routines are available in all precision types: single precision, double precision, complexm, and double complex precision. See the `mkl_scalapack.h` header file for C declarations of ScaLAPACK routines.

NOTE

ScaLAPACK routines are provided only with Intel® MKL versions for Linux* and Windows* OSs.

Sections in this chapter include descriptions of ScaLAPACK [computational routines](#) that perform distinct computational tasks, as well as [driver routines](#) for solving standard types of problems in one call.

Generally, ScaLAPACK runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of BLAS optimized for the target architecture. Intel MKL version of ScaLAPACK is optimized for Intel® processors. For the detailed system and environment requirements, see *Intel® MKL Release Notes* and *Intel® MKL User's Guide*.

For full reference on ScaLAPACK routines and related information, see [\[SLUG\]](#).

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Overview

The model of the computing environment for ScaLAPACK is represented as a one-dimensional array of processes (for operations on band or tridiagonal matrices) or also a two-dimensional process grid (for operations on dense matrices). To use ScaLAPACK, all global matrices or vectors should be distributed on this array or grid prior to calling the ScaLAPACK routines.

ScaLAPACK uses the two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, and also gives the opportunity to use BLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global matrix (array) and its corresponding process and memory location is contained in the array called the *array descriptor* associated with each global matrix.

Let A be a two-dimensional block cyclicly distributed matrix with the array descriptor array $desca$. The meaning of each array descriptor element depends on the type of the matrix A . The tables [Array descriptor for dense matrices](#) and [Array descriptor for narrow-band and tridiagonal matrices](#) describe the meaning of each element for the different types of matrices.

Array descriptor for dense matrices ($dlen=9$)

Element Name	Stored in	Description	Element Index Number
$dtype_a$	$desca(dtype_)$	Descriptor type (=1 for dense matrices).	1
$ctxt_a$	$desca(ctxt_)$	BLACS context handle for the process grid.	2
m_a	$desca(m_)$	Number of rows in the global matrix A .	3
n_a	$desca(n_)$	Number of columns in the global matrix A .	4
mb_a	$desca(mb_)$	Row blocking factor.	5
nb_a	$desca(nb_)$	Column blocking factor.	6
$rsrc_a$	$desca(rsrc_)$	Process row over which the first row of the global matrix A is distributed.	7
$csrc_a$	$desca(csrc_)$	Process column over which the first column of the global matrix A is distributed.	8
lId_a	$desca(lId_)$	Leading dimension of the local matrix A .	9

Array descriptor for narrow-band and tridiagonal matrices ($dlen=7$)

Element Name	Stored in	Description	Element Index Number
$dtype_a$	$desca(dtype_)$	Descriptor type	1
		<ul style="list-style-type: none"> • $dtype_a=501$: 1-by-P grid, • $dtype_a=502$: P-by-1 grid. 	
$ctxt_a$	$desca(ctxt_)$	BLACS context handle indicating the BLACS process grid over which the global matrix A is distributed. The context itself is global, but the handle (the integer value) can vary.	2
n_a	$desca(n_)$	The size of the matrix dimension being distributed.	3
nb_a	$desca(nb_)$	The blocking factor used to distribute the distributed dimension of the matrix A .	4
src_a	$desca(src_)$	The process row or column over which the first row or column of the matrix A is distributed.	5
lId_a	$desca(lId_)$	The leading dimension of the local matrix storing the local blocks of the distributed matrix A . The minimum value of lId_a depends on $dtype_a$. <ul style="list-style-type: none"> • $dtype_a=501$: $lId_a \geq \max(\text{size of undistributed dimension}, 1)$, • $dtype_a=502$: $lId_a \geq \max(nb_a, 1)$. 	6
Not applicable		Reserved for future use.	7

Similar notations are used for different matrices. For example: lId_b is the leading dimension of the local matrix storing the local blocks of the distributed matrix B and $dtype_z$ is the type of the global matrix Z .

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by $LOC_r()$ and $LOC_c()$, respectively. To compute these numbers, you can use the ScaLAPACK tool routine `numroc`.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix $\text{sub}(A)$ of the global matrix A defined by the following 6 values (for dense matrices):

m	The number of rows of $\text{sub}(A)$
n	The number of columns of $\text{sub}(A)$

<i>a</i>	A pointer to the local matrix containing the entire global matrix A
<i>ia</i>	The row index of sub(<i>A</i>) in the global matrix <i>A</i>
<i>ja</i>	The column index of sub(<i>A</i>) in the global matrix <i>A</i>
<i>desca</i>	The array descriptor for the global matrix <i>A</i>

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Routine Naming Conventions

For each routine introduced in this chapter, you can use the ScalAPACK name. The naming convention for ScalAPACK routines is similar to that used for LAPACK routines. A general rule is that each routine name in ScalAPACK, which has an LAPACK equivalent, is simply the LAPACK name prefixed by initial letter *p*.

ScalAPACK names have the structure *p?yyzzz* or *p?yyzz*, which is described below.

The initial letter *p* is a distinctive prefix of ScalAPACK routines and is present in each such routine.

The second symbol *?* indicates the data type:

<i>s</i>	real, single precision
<i>d</i>	real, double precision
<i>c</i>	complex, single precision
<i>z</i>	complex, double precision

The second and third letters *yy* indicate the matrix type as:

<i>ge</i>	general
<i>gb</i>	general band
<i>gg</i>	a pair of general matrices (for a generalized problem)
<i>dt</i>	general tridiagonal (diagonally dominant-like)
<i>db</i>	general band (diagonally dominant-like)
<i>po</i>	symmetric or Hermitian positive-definite
<i>pb</i>	symmetric or Hermitian positive-definite band
<i>pt</i>	symmetric or Hermitian positive-definite tridiagonal
<i>sy</i>	symmetric

st	symmetric tridiagonal (real)
he	Hermitian
or	orthogonal
tr	triangular (or quasi-triangular)
tz	trapezoidal
un	unitary

For computational routines, the last three letters **zzz** indicate the computation performed and have the same meaning as for LAPACK routines.

For driver routines, the last two letters **zz** or three letters **zzz** have the following meaning:

sv	a <i>simple</i> driver for solving a linear system
svx	an <i>expert</i> driver for solving a linear system
ls	a driver for solving a linear least squares problem
ev	a simple driver for solving a symmetric eigenvalue problem
evd	a simple driver for solving an eigenvalue problem using a divide and conquer algorithm
evx	an expert driver for solving a symmetric eigenvalue problem
svd	a driver for computing a singular value decomposition
gvx	an expert driver for solving a generalized symmetric definite eigenvalue problem

Simple driver here means that the driver just solves the general problem, whereas an *expert* driver is more versatile and can also optionally perform some related computations (such, for example, as refining the solution and computing error bounds after the linear system is solved).

Computational Routines

In the sections that follow, the descriptions of ScaLAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

- [Solving Systems of Linear Equations](#)
- [Orthogonal Factorizations and LLS Problems](#)
- [Symmetric Eigenproblems](#)
- [Nonsymmetric Eigenproblems](#)
- [Singular Value Decomposition](#)
- [Generalized Symmetric-Definite Eigenproblems](#)

See also the respective [driver routines](#).

Linear Equations

ScaLAPACK supports routines for the systems of equations with the following types of matrices:

- general
- general banded

- general diagonally dominant-like banded (including general tridiagonal)
- symmetric or Hermitian positive-definite
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian positive-definite tridiagonal

A *diagonally dominant-like* matrix is defined as a matrix for which it is known in advance that pivoting is not required in the *LU* factorization of this matrix.

For the above matrix types, the library includes routines for performing the following computations: *factoring* the matrix; *equilibrating* the matrix; *solving* a system of linear equations; *estimating the condition number* of a matrix; *refining* the solution of linear equations and computing its error bounds; *inverting* the matrix. Note that for some of the listed matrix types only part of the computational routines are provided (for example, routines that refine the solution are not provided for band or tridiagonal matrices). See [Table "Computational Routines for Systems of Linear Equations"](#) for full list of available routines.

To solve a particular problem, you can either call two or more computational routines or call a corresponding *driver routine* that combines several tasks in one call. Thus, to solve a system of linear equations with a general matrix, you can first call `p?getrf`(*LU* factorization) and then `p?getrs`(computing the solution).

Then, you might wish to call `p?gerfs` to refine the solution and get the error bounds. Alternatively, you can just use the driver routine `p?gesvx` which performs all these tasks in one call.

[Table "Computational Routines for Systems of Linear Equations"](#) lists the ScaLAPACK computational routines for factorizing, equilibrating, and inverting matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error.

Computational Routines for Systems of Linear Equations

Matrix type, storage scheme	Factorize matrix	Equilibrate matrix	Solve system	Condition number	Estimate error	Invert matrix
general (partial pivoting)	<code>p?getrf</code>	<code>p?geequ</code>	<code>p?getrs</code>	<code>p?gecon</code>	<code>p?gerfs</code>	<code>p?getri</code>
general band (partial pivoting)	<code>p?gbtrf</code>		<code>p?gbtrs</code>			
general band (no pivoting)	<code>p?dbtrf</code>		<code>p?dbtrs</code>			
general tridiagonal (no pivoting)	<code>p?dttrf</code>		<code>p?dttrs</code>			
symmetric/Hermitian positive-definite	<code>p?potrf</code>	<code>p?poequ</code>	<code>p?potrs</code>	<code>p?pocon</code>	<code>p?porfs</code>	<code>p?potri</code>
symmetric/Hermitian positive-definite, band	<code>p?pbtrf</code>		<code>p?pbtrs</code>			
symmetric/Hermitian positive-definite, tridiagonal	<code>p?pttrf</code>		<code>p?pttrs</code>			
triangular			<code>p?trtrs</code>	<code>p?trcon</code>	<code>p?trrfs</code>	<code>p?trtri</code>

In this table ? stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex).

Routines for Matrix Factorization

This section describes the ScaLAPACK routines for matrix factorization. The following factorizations are supported:

- LU factorization of general matrices
- LU factorization of diagonally dominant-like matrices
- Cholesky factorization of real symmetric or complex Hermitian positive-definite matrices

You can compute the factorizations using full and band storage of matrices.

`p?getrf`

Computes the LU factorization of a general m-by-n distributed matrix.

Syntax

Fortran:

```
call psgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pdgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pcgetrf(m, n, a, ia, ja, desca, ipiv, info)
call pzgetrf(m, n, a, ia, ja, desca, ipiv, info)
```

C:

```
void psgetrf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *ipiv , MKL_INT *info );
void pdgetrf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
void pcgetrf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
void pzgetrf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?getrf routine forms the LU factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$$A = P \cdot L \cdot U$$

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$). L and U are stored in $\text{sub}(A)$.

The routine uses partial pivoting, with row interchanges.

Input Parameters

m (global) INTEGER. The number of rows in the distributed submatrix $\text{sub}(A)$; $n \geq 0$.

n (global) INTEGER. The number of columns in the distributed submatrix $\text{sub}(A)$; $n \geq 0$.

a (local)

```
REAL for psgetrf
DOUBLE PRECISION for pdgetrf
COMPLEX for pcgetrf
DOUBLE COMPLEX for pzgetrf.
```

Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc(ja+n-1)*).

Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix $A(ia:ia+n-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	Overwritten by local pieces of the factors <i>L</i> and <i>U</i> from the factorization $A = P*L*U$. The unit diagonal elements of <i>L</i> are not stored.
<i>ipiv</i>	(local) INTEGER array. The dimension of <i>ipiv</i> is (<i>LOCr(m_a) + mb_a</i>). This array contains the pivoting information: local row <i>i</i> was interchanged with global row <i>ipiv(i)</i> . This array is tied to the distributed matrix <i>A</i> .
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . If <i>info</i> = <i>i</i> , u_{ii} is 0. The factorization has been completed, but the factor <i>U</i> is exactly singular. Division by zero will occur if you use the factor <i>U</i> for solving a system of linear equations.

p?gbtrf

Computes the LU factorization of a general *n*-by-*n* banded distributed matrix.

Syntax

Fortran:

```
call psgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pdgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pcgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
call pzgbtrf(n, bwl, bwu, a, ja, desca, ipiv, af, laf, work, lwork, info)
```

C:

```
void psgbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , float *a , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , float *af , MKL_INT *laf , float *work , MKL_INT
*lwork , MKL_INT *info );
void pdgbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , double *a , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , double *af , MKL_INT *laf , double *work , MKL_INT
*lwork , MKL_INT *info );
void pcgbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_Complex8 *a , MKL_INT
*ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8
*work , MKL_INT *lwork , MKL_INT *info );
```

```
void pzgbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_Complex16 *a , MKL_INT
*ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *af , MKL_INT *laf ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?gbtrf` routine computes the *LU* factorization of a general n -by- n real/complex banded distributed matrix $A(1:n, ja:ja+n-1)$ using partial pivoting with row interchanges.

The resulting factorization is not the same factorization as returned from the LAPACK routine `?gbtrf`. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form

$$A(1:n, ja:ja+n-1) = P \cdot L \cdot U \cdot Q$$

where P and Q are permutation matrices, and L and U are banded lower and upper triangular matrices, respectively. The matrix Q represents reordering of columns for the sake of parallelism, while P represents reordering of rows for numerical stability using classic partial pivoting.

Input Parameters

<i>n</i>	(global) INTEGER. The number of rows and columns in the distributed submatrix $A(1:n, ja:ja+n-1)$; $n \geq 0$.
<i>bwl</i>	(global) INTEGER. The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).
<i>bwu</i>	(global) INTEGER. The number of super-diagonals within the band of A ($0 \leq bwu \leq n-1$).
<i>a</i>	(local) REAL for <code>psgbtrf</code> DOUBLE PRECISION for <code>pdgbtrf</code> COMPLEX for <code>pcgbtrf</code> DOUBLE COMPLEX for <code>pzgbtrf</code> . Pointer into the local memory to an array of local dimension (<i>lld_a</i> , $LOC_C(ja+n-1)$) where $lld_a \geq 2*bwl + 2*bwu + 1$. Contains the local pieces of the n -by- n distributed banded matrix $A(1:n, ja:ja+n-1)$ to be factored.
<i>ja</i>	(global) INTEGER. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix A . If $desca(dtype_) = 501$, then $dlen_ \geq 7$;

else if $\text{desca}(\text{dtype}_\perp) = 1$, then $\text{dlen}_\perp \geq 9$.

laf

(local) INTEGER. The dimension of the array af .

Must be $\text{laf} \geq (\text{NB}+\text{bwu}) * (\text{bwl}+\text{bwu}) + 6 * (\text{bwl}+\text{bwu}) * (\text{bwl}+2 * \text{bwu})$.

If laf is not large enough, an error code will be returned and the minimum acceptable size will be returned in $\text{af}(1)$.

work

(local) Same type as a . Workspace array of dimension lwork .

lwork

(local or global) INTEGER. The size of the work array ($\text{lwork} \geq 1$). If lwork is too small, the minimal acceptable size will be returned in $\text{work}(1)$ and an error code is returned.

Output Parameters

a

On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

ipiv

(local) INTEGER array.

The dimension of ipiv must be $\geq \text{desca}(\text{NB})$.

Contains pivot indices for local factorizations. Note that you *should not alter* the contents of this array between factorization and solve.

af

(local)

REAL for psgbtrf

DOUBLE PRECISION for pdgbtrf

COMPLEX for pcgbtrf

DOUBLE COMPLEX for pzgbtrf .

Array, dimension (laf).

Auxiliary Fillin space. Fillin is created during the factorization routine p?gbtrf and this is stored in af .

Note that if a linear system is to be solved using p?gbtrs after the factorization routine, af must not be altered after the factorization.

$\text{work}(1)$

On exit, $\text{work}(1)$ contains the minimum value of lwork required for optimum performance.

info

(global) INTEGER.

If $\text{info}=0$, the execution is successful.

$\text{info} < 0$:

If the i th argument is an array and the j th entry had an illegal value, then $\text{info} = -(i*100+j)$; if the i th argument is a scalar and had an illegal value, then $\text{info} = -i$.

$\text{info} > 0$:

If $\text{info} = k \leq \text{NPROCS}$, the submatrix stored on processor info and factored locally was not nonsingular, and the factorization was not completed. If $\text{info} = k > \text{NPROCS}$, the submatrix stored on processor info-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dbtrf

Computes the LU factorization of a n-by-n diagonally dominant-like banded distributed matrix.

Syntax

Fortran:

```
call psdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pddbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pcdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
call pzdbtrf(n, bwl, bwu, a, ja, desca, af, laf, work, lwork, info)
```

C:

```
void psdbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , float *a , MKL_INT *ja ,
MKL_INT *desca , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT
*info );
void pddbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , double *a , MKL_INT *ja ,
MKL_INT *desca , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT
*info );
void pcdbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_Complex8 *a , MKL_INT
*ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
void pzdbtrf (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_Complex16 *a , MKL_INT
*ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?dbtrf` routine computes the LU factorization of a n -by- n real/complex diagonally dominant-like banded distributed matrix $A(1:n, ja:ja+n-1)$ without pivoting.

Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

Input Parameters

n	(global) INTEGER. The number of rows and columns in the distributed submatrix $A(1:n, ja:ja+n-1)$; $n \geq 0$.
bwl	(global) INTEGER. The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).

<i>bwu</i>	(global) INTEGER. The number of super-diagonals within the band of <i>A</i> $(0 \leq bwu \leq n-1)$.
<i>a</i>	(local) REAL for psdbtrf DOUBLE PRECISION for pdhbtrf COMPLEX for pcdbtrf DOUBLE COMPLEX for pzdbtrf. Pointer into the local memory to an array of local dimension $(lld_a, LOCc(ja+n-1))$. Contains the local pieces of the n -by- n distributed banded matrix $A(1:n, ja:ja+n-1)$ to be factored.
<i>ja</i>	(global) INTEGER. The index in the global array <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca(dtype_)</i> = 501, then <i>dlen_</i> ≥ 7 ; else if <i>desca(dtype_)</i> = 1, then <i>dlen_</i> ≥ 9 .
<i>laf</i>	(local) INTEGER. The dimension of the array <i>af</i> . Must be <i>laf</i> $\geq NB*(bwl+bwu)+6*(\max(bwl,bwu))^2$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af(1)</i> .
<i>work</i>	(local) Same type as <i>a</i> . Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the <i>work</i> array, must be <i>lwork</i> $\geq (\max(bwl,bwu))^2$. If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work(1)</i> and an error code is returned.

Output Parameters

<i>a</i>	On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.
<i>af</i>	(local) REAL for psdbtrf DOUBLE PRECISION for pdhbtrf COMPLEX for pcdbtrf DOUBLE COMPLEX for pzdbtrf. Array, dimension (<i>laf</i>). Auxiliary Fillin space. Fillin is created during the factorization routine <i>p?dbtrf</i> and this is stored in <i>af</i> .

Note that if a linear system is to be solved using `p?dbtrs` after the factorization routine, `af` must not be altered after the factorization.

`work(1)`

On exit, `work(1)` contains the minimum value of `lwork` required for optimum performance.

`info`

(global) INTEGER.

If `info=0`, the execution is successful.

`info < 0`:

If the `i`-th argument is an array and the `j`-th entry had an illegal value, then `info = -(i*100+j)`; if the `i`-th argument is a scalar and had an illegal value, then `info = -i`. `info > 0`:

If `info = k ≤ NPROCS`, the submatrix stored on processor `info` and factored locally was not diagonally dominant-like, and the factorization was not completed. If `info = k > NPROCS`, the submatrix stored on processor `info-NPROCS` representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dttrf

Computes the LU factorization of a diagonally dominant-like tridiagonal distributed matrix.

Syntax

Fortran:

```
call psdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pddttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pcdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
call pzdttrf(n, dl, d, du, ja, desca, af, laf, work, lwork, info)
```

C:

```
void psdttrf (MKL_INT *n , float *dl , float *d , float *du , MKL_INT *ja , MKL_INT
*idesca , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );
void pddttrf (MKL_INT *n , double *dl , double *d , double *du , MKL_INT *ja , MKL_INT
*idesca , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );
void pcdttrf (MKL_INT *n , MKL_Complex8 *dl , MKL_Complex8 *d , MKL_Complex8 *du ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );
void pzdttrf (MKL_INT *n , MKL_Complex16 *dl , MKL_Complex16 *d , MKL_Complex16 *du ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16
*work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: `mkl_scalapack.h`

Description

The `p?dttrf` routine computes the LU factorization of an n -by- n real/complex diagonally dominant-like tridiagonal distributed matrix $A(1:n, ja:ja+n-1)$ without pivoting for stability.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$$A(1:n, ja:ja+n-1) = P \cdot L \cdot U \cdot P^T,$$

where P is a permutation matrix, and L and U are banded lower and upper triangular matrices, respectively.

Input Parameters

n	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$ ($n \geq 0$).
dl, d, du	(local) REAL for pspttrf DOUBLE PRECISON for pdpttrf COMPLEX for pcpttrf DOUBLE COMPLEX for pzpttrf. Pointers to the local arrays of dimension $(desca(nb_))$ each. On entry, the array dl contains the local part of the global vector storing the subdiagonal elements of the matrix. Globally, $dl(1)$ is not referenced, and dl must be aligned with d . On entry, the array d contains the local part of the global vector storing the diagonal elements of the matrix. On entry, the array du contains the local part of the global vector storing the super-diagonal elements of the matrix. $du(n)$ is not referenced, and du must be aligned with d .
ja	(global) INTEGER. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
$desca$	(global and local) INTEGER array, dimension $(dlen_)$. The array descriptor for the distributed matrix A . If $desca(dtype_)$ = 501, then $dlen_ \geq 7$; else if $desca(dtype_)$ = 1, then $dlen_ \geq 9$.
laf	(local) INTEGER. The dimension of the array af . Must be $laf \geq 2*(NB+2)$. If laf is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af(1)$.
$work$	(local) Same type as d . Workspace array of dimension $lwork$.
$lwork$	(local or global) INTEGER. The size of the $work$ array, must be at least $lwork \geq 8*NPCOL$.

Output Parameters

dl, d, du	On exit, overwritten by the information containing the factors of the matrix.
af	(local)

REAL for psdttrf
 DOUBLE PRECISION for pdattrf
 COMPLEX for pcdttrf
 DOUBLE COMPLEX for pzdttrf.
 Array, dimension (*laf*).

Auxiliary Fillin space. Fillin is created during the factorization routine p?dttrf and this is stored in *af*.

Note that if a linear system is to be solved using [p?dttrs](#) after the factorization routine, *af* must not be altered.

work(1) On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.

If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*) ; if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:

If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not diagonally dominant-like, and the factorization was not completed. If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?potrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite distributed matrix.

Syntax

Fortran:

```
call pspotrf(uplo, n, a, ia, ja, desca, info)
call pdpotrf(uplo, n, a, ia, ja, desca, info)
call pcpotrf(uplo, n, a, ia, ja, desca, info)
call pzpotrf(uplo, n, a, ia, ja, desca, info)
```

C:

```
void pspotrf (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
void pdpotrf (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
void pcpotrf (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
```

```
void pzpotrf (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?potrf` routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive-definite distributed n -by- n matrix $A(ia:ia+n-1, ja:ja+n-1)$, denoted below as $\text{sub}(A)$.

The factorization has the form

$\text{sub}(A) = U^H * U$ if $\text{uplo} = 'U'$, or

$\text{sub}(A) = L * L^H$ if $\text{uplo} = 'L'$

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

<code>uplo</code>	(global) CHARACTER*1. Indicates whether the upper or lower triangular part of $\text{sub}(A)$ is stored. Must be ' <code>U</code> ' or ' <code>L</code> '. If <code>uplo</code> = ' <code>U</code> ', the array <code>a</code> stores the upper triangular part of the matrix <code>sub(A)</code> that is factored as $U^H * U$. If <code>uplo</code> = ' <code>L</code> ', the array <code>a</code> stores the lower triangular part of the matrix <code>sub(A)</code> that is factored as $L * L^H$.
<code>n</code>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<code>a</code>	(local) REAL for <code>pspotrf</code> DOUBLE PRECISON for <code>pdpotrf</code> COMPLEX for <code>pcpotrf</code> DOUBLE COMPLEX for <code>pzpotrf</code> . Pointer into the local memory to an array of dimension $(\text{lld}_a, \text{LOCc}(ja + n - 1))$. On entry, this array contains the local pieces of the n -by- n symmetric/ Hermitian distributed matrix $\text{sub}(A)$ to be factored. Depending on <code>uplo</code> , the array <code>a</code> contains either the upper or the lower triangular part of the matrix $\text{sub}(A)$ (see <code>uplo</code>).
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<code>desca</code>	(global and local) INTEGER array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix A .

Output Parameters

a The upper or lower triangular part of *a* is overwritten by the Cholesky factor *U* or *L*, as specified by *uplo*.

info (global) INTEGER.
 If *info*=0, the execution is successful;
info < 0: if the *i*-th argument is an array, and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
 If *info* = *k* > 0, the leading minor of order *k*, *A*(*ia*:*ia+k-1*, *ja*:*ja+k-1*), is not positive-definite, and the factorization could not be completed.

p?pbtrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite banded distributed matrix.

Syntax

Fortran:

```
call pspbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pdpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pcpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
call pzpbtrf(uplo, n, bw, a, ja, desca, af, laf, work, lwork, info)
```

C:

```
void pspbtrf (char *uplo , MKL_INT *n , MKL_INT *bw , float *a , MKL_INT *ja , MKL_INT
*desca , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );
void pdpbtrf (char *uplo , MKL_INT *n , MKL_INT *bw , double *a , MKL_INT *ja ,
MKL_INT *desca , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT
*info );
void pcpbtrf (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_Complex8 *a , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
void pzpbtrf (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_Complex16 *a , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?pbtrf routine computes the Cholesky factorization of an *n*-by-*n* real symmetric or complex Hermitian positive-definite banded distributed matrix *A*(1:*n*, *ja*:*ja+n-1*).

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$A(1:n, ja:ja+n-1) = P * U^H * U * P^T$, if $\text{uplo} = \text{'U'}$, or

$A(1:n, ja:ja+n-1) = P * L * L^H * P^T$, if $\text{uplo} = \text{'L'}$,

where P is a permutation matrix and U and L are banded upper and lower triangular matrices, respectively.

Input Parameters

uplo

(global) CHARACTER*1. Must be '*U*' or '*L*'.

If *uplo* = '*U*', upper triangle of $A(1:n, ja:ja+n-1)$ is stored;

If *uplo* = '*L*', lower triangle of $A(1:n, ja:ja+n-1)$ is stored.

n

(global) INTEGER. The order of the distributed submatrix $A(1:n, ja:ja+n-1)$.

($n \geq 0$).

bw

(global) INTEGER.

The number of superdiagonals of the distributed matrix if *uplo* = '*U*', or the number of subdiagonals if *uplo* = '*L*' ($bw \geq 0$).

a

(local)

REAL for pspbtrf

DOUBLE PRECISON for pdpbtrf

COMPLEX for pcpbtrf

DOUBLE COMPLEX for pzpbtrf.

Pointer into the local memory to an array of dimension

(*lld_a*, *LOCc(ja+n-1)*).

On entry, this array contains the local pieces of the upper or lower triangle of the symmetric/Hermitian band distributed matrix $A(1:n, ja:ja+n-1)$ to be factored.

ja

(global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

If *desca(dtype_)* = 501, then *dlen_* ≥ 7 ;

else if *desca(dtype_)* = 1, then *dlen_* ≥ 9 .

laf

(local) INTEGER. The dimension of the array *af*.

Must be *laf* $\geq (\text{NB} + 2 * bw) * bw$.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af(1)*.

work

(local) Same type as *a*. Workspace array of dimension *lwork*.

lwork

(local or global) INTEGER. The size of the *work* array, must be *lwork* $\geq bw^2$.

Output Parameters

a On exit, if *info*=0, contains the permuted triangular factor *U* or *L* from the Cholesky factorization of the band matrix $A(1:n, ja:ja+n-1)$, as specified by *uplo*.

af (local)
 REAL for pspbtrf
 DOUBLE PRECISON for pdpbtrf
 COMPLEX for pcpbtrf
 DOUBLE COMPLEX for pzpbtrf.

Array, dimension (*laf*). Auxiliary Fillin space. Fillin is created during the factorization routine *p?pbtrf* and this is stored in *af*. Note that if a linear system is to be solved using *p?pbtrs* after the factorization routine, *af* must not be altered.

work(1) On exit, *work(1)* contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
 If *info*=0, the execution is successful.
info < 0:
 If the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*) ; if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*.
info>0:
 If *info* = *k* ≤ NPROCS, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.
 If *info* = *k* > NPROCS, the submatrix stored on processor *info*-NPROCS representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?pttrf

Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite tridiagonal distributed matrix.

Syntax

Fortran:

```
call pspttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pdpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pcpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
call pzpttrf(n, d, e, ja, desca, af, laf, work, lwork, info)
```

C:

```

void pspttrf (MKL_INT *n , float *d , float *e , MKL_INT *ja , MKL_INT *desca , float
*af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pdpttrf (MKL_INT *n , double *d , double *e , MKL_INT *ja , MKL_INT *desca ,
double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcpttrf (MKL_INT *n , float *d , MKL_Complex8 *e , MKL_INT *ja , MKL_INT *desca ,
MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzpttrf (MKL_INT *n , double *d , MKL_Complex16 *e , MKL_INT *ja , MKL_INT
*desca , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT *lwork ,
MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The p?pttrf routine computes the Cholesky factorization of an n -by- n real symmetric or complex hermitian positive-definite tridiagonal distributed matrix $A(1:n, ja:ja+n-1)$.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$$A(1:n, ja:ja+n-1) = P \cdot L \cdot D \cdot L^H \cdot P^T, \text{ or}$$

$$A(1:n, ja:ja+n-1) = P \cdot U^H \cdot D \cdot U \cdot P^T,$$

where P is a permutation matrix, and U and L are tridiagonal upper and lower triangular matrices, respectively.

Input Parameters

n (global) INTEGER. The order of the distributed submatrix $A(1:n,$
 $ja:ja+n-1)$
 $(n \geq 0)$.

d, e (local)
REAL for pspttrf
DOUBLE PRECISON for pdpttrf
COMPLEX for pcpttrf
DOUBLE COMPLEX for pzpttrf.
Pointers into the local memory to arrays of dimension (*desca(nb_)*) each.
On entry, the array *d* contains the local part of the global vector storing the main diagonal of the distributed matrix *A*.
On entry, the array *e* contains the local part of the global vector storing the upper diagonal of the distributed matrix *A*.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca(dtype_)</i> = 501, then <i>dlen_</i> ≥ 7 ; else if <i>desca(dtype_)</i> = 1, then <i>dlen_</i> ≥ 9 .
<i>laf</i>	(local) INTEGER. The dimension of the array <i>af</i> . Must be <i>laf</i> $\geq \text{NB}+2$. If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>work</i>	(local) Same type as <i>d</i> and <i>e</i> . Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the <i>work</i> array, must be at least <i>lwork</i> $\geq 8 * \text{NPCOL}$.

Output Parameters

<i>d, e</i>	On exit, overwritten by the details of the factorization.
<i>af</i>	(local) REAL for pspttrf DOUBLE PRECISION for pdpttrf COMPLEX for pcpttrf DOUBLE COMPLEX for pzpttrf. Array, dimension (<i>laf</i>). Auxiliary Fillin space. Fillin is created during the factorization routine p?pttrf and this is stored in <i>af</i> . Note that if a linear system is to be solved using p?pttrs after the factorization routine, <i>af</i> must not be altered.
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$. <i>info</i> > 0: If <i>info</i> = <i>k</i> $\leq \text{NPROCS}$, the submatrix stored on processor <i>info</i> and factored locally was not positive definite, and the factorization was not completed. If <i>info</i> = <i>k</i> $> \text{NPROCS}$, the submatrix stored on processor <i>info-NPROCS</i> representing interactions with other processors was not nonsingular, and the factorization was not completed.

Routines for Solving Systems of Linear Equations

This section describes the ScalAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see [Routines for Matrix Factorization](#) in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

p?getrs

Solves a system of distributed linear equations with a general square matrix, using the LU factorization computed by p?getrf.

Syntax

Fortran:

```
call psgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pdgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pcgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pzgetrs(trans, n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

C:

```
void psgetrs (char *trans , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , float *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );
void pdgetrs (char *trans , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , double *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );
void pcgetrs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
void pzgetrs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?getrs routine solves a system of distributed linear equations with a general n -by- n distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the LU factorization computed by p?getrf.

The system has one of the following forms specified by *trans*:

$\text{sub}(A) * X = \text{sub}(B)$ (no transpose),
 $\text{sub}(A)^T * X = \text{sub}(B)$ (transpose),
 $\text{sub}(A)^H * X = \text{sub}(B)$ (conjugate transpose),
where $\text{sub}(B) = B(ib:ib+n-1, jb:jb+nrhs-1)$.

Before calling this routine, you must call p?getrf to compute the LU factorization of $\text{sub}(A)$.

Input Parameters

<i>trans</i>	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $\text{sub}(A)^*X = \text{sub}(B)$ is solved for X . If <i>trans</i> = 'T', then $\text{sub}(A)^T*X = \text{sub}(B)$ is solved for X . If <i>trans</i> = 'C', then $\text{sub}(A)^H*X = \text{sub}(B)$ is solved for X .
<i>n</i>	(global) INTEGER. The number of linear equations; the order of the submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>a, b</i>	(global) REAL for psgetrs DOUBLE PRECISION for pdgetrs COMPLEX for pcgetrs DOUBLE COMPLEX for pzgetrs. Pointers into the local memory to arrays of local dimension $a(lld_a, LOCr(ja+n-1))$ and $b(lld_b, LOCr(jb+nrhs-1))$, respectively. On entry, the array <i>a</i> contains the local pieces of the factors <i>L</i> and <i>U</i> from the factorization $\text{sub}(A) = P^*L^*U$; the unit diagonal elements of <i>L</i> are not stored. On entry, the array <i>b</i> contains the right hand sides $\text{sub}(B)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>ipiv</i>	(local) INTEGER array. The dimension of <i>ipiv</i> is $(LOCr(m_a) + mb_a)$. This array contains the pivoting information: local row <i>i</i> of the matrix was interchanged with the global row <i>ipiv(i)</i> . This array is tied to the distributed matrix <i>A</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>b</i>	On exit, overwritten by the solution distributed matrix <i>X</i> .
----------	--

info INTEGER. If *info*=0, the execution is successful. *info* < 0:
 If the *i*-th argument is an array and the *j*-th entry had an illegal value, then
 $info = -(i*100+j)$; if the *i*-th argument is a scalar and had an illegal
 value, then *info* = *i*.

p?gbtrs

Solves a system of distributed linear equations with a general band matrix, using the LU factorization computed by p?gbtrf.

Syntax

Fortran:

```
call psgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf,
work, lwork, info)

call pdgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf,
work, lwork, info)

call pcgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf,
work, lwork, info)

call pzgbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, af, laf,
work, lwork, info)
```

C:

```
void psgbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
float *a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , float *b , MKL_INT *ib ,
MKL_INT *descb , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT
*info );

void pdgbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
double *a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , double *b , MKL_INT *ib ,
MKL_INT *descb , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT
*info );

void pcgbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzgbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16
*work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?gbtrs routine solves a system of distributed linear equations with a general band distributed matrix $\text{sub}(A) = A(1:n, ja:ja+n-1)$ using the LU factorization computed by p?gbtrf.

The system has one of the following forms specified by *trans*:

$\text{sub}(A) * X = \text{sub}(B)$ (no transpose),

$\text{sub}(A)^T X = \text{sub}(B)$ (transpose),
 $\text{sub}(A)^H X = \text{sub}(B)$ (conjugate transpose),
where $\text{sub}(B) = B(ib:ib+n-1, 1:nrhs)$.

Before calling this routine, you must call [p?gbtrf](#) to compute the LU factorization of $\text{sub}(A)$.

Input Parameters

<i>trans</i>	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'. Indicates the form of the equations: If <i>trans</i> = 'N', then $\text{sub}(A)^* X = \text{sub}(B)$ is solved for X . If <i>trans</i> = 'T', then $\text{sub}(A)^T X = \text{sub}(B)$ is solved for X . If <i>trans</i> = 'C', then $\text{sub}(A)^H X = \text{sub}(B)$ is solved for X .
<i>n</i>	(global) INTEGER. The number of linear equations; the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>bwl</i>	(global) INTEGER. The number of sub-diagonals within the band of A ($0 \leq bwl \leq n-1$).
<i>bwu</i>	(global) INTEGER. The number of super-diagonals within the band of A ($0 \leq bwu \leq n-1$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>a, b</i>	(global) REAL for psgbtrs DOUBLE PRECISION for pdgbtrs COMPLEX for pcgbtrs DOUBLE COMPLEX for pzgbtrs. Pointers into the local memory to arrays of local dimension $a(1ld_a, LOC_c(ja+n-1))$ and $b(1ld_b, LOC_c(nrhs))$, respectively. The array <i>a</i> contains details of the LU factorization of the distributed band matrix <i>A</i> . On entry, the array <i>b</i> contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.
<i>ja</i>	(global) INTEGER. The index in the global array <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca(dtype_)</i> = 501, then <i>dlen_</i> ≥ 7 ; else if <i>desca(dtype_)</i> = 1, then <i>dlen_</i> ≥ 9 .

<i>ib</i>	(global) INTEGER. The index in the global array <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca(dtype_)</i> = 502 , then <i>dlen_</i> \geq 7; else if <i>desca(dtype_)</i> = 1, then <i>dlen_</i> \geq 9.
<i>laf</i>	(local) INTEGER. The dimension of the array <i>af</i> . Must be <i>laf</i> \geq NB*(<i>bwl</i> + <i>bwu</i>)+6* <i>(bwl</i> + <i>bwu</i>)*(<i>bwl</i> +2* <i>bwu</i>). If <i>laf</i> is not large enough, an error code will be returned and the minimum acceptable size will be returned in <i>af</i> (1).
<i>work</i>	(local) Same type as <i>a</i> . Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. The size of the <i>work</i> array, must be at least <i>lwork</i> \geq <i>nrhs</i> * (NB+2 * <i>bwl</i> +4 * <i>bwu</i>).

Output Parameters

<i>ipiv</i>	(local) INTEGER array. The dimension of <i>ipiv</i> must be \geq <i>desca</i> (NB) . Contains pivot indices for local factorizations. Note that you should not alter the contents of this array between factorization and solve.
<i>b</i>	On exit, overwritten by the local pieces of the solution distributed matrix <i>X</i> .
<i>af</i>	(local) REAL for psgbtrs DOUBLE PRECISION for pdgbtrs COMPLEX for pcgbtrs DOUBLE COMPLEX for pzgbtrs. Array, dimension (<i>laf</i>). Auxiliary Fillin space. Fillin is created during the factorization routine p?gbtrf and this is stored in <i>af</i> . Note that if a linear system is to be solved using p?gbtrs after the factorization routine, <i>af</i> must not be altered after the factorization.
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> th entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>) ; if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?dbtrs

Solves a system of linear equations with a diagonally dominant-like banded distributed matrix using the factorization computed by p?dbtrf.

Syntax

FORTRAN 77:

```
call psdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)

call pddbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)

call pcdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)

call pzdbtrs(trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

C:

```
void psdbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
float *a , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb ,
float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pddbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
double *a , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb ,
double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcdbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzdbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?dbtrs routine solves for X one of the systems of equations:

```
sub(A)*X = sub(B),
(sub(A))T*X = sub(B), or
(sub(A))H*X = sub(B),
```

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is a diagonally dominant-like banded distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses the LU factorization computed by p?dbtrf.

Input Parameters

<i>trans</i>	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
--------------	--

Indicates the form of the equations:

If $\text{trans} = \text{'N'}$, then $\text{sub}(A) * X = \text{sub}(B)$ is solved for X .

If $\text{trans} = \text{'T'}$, then $(\text{sub}(A))^T * X = \text{sub}(B)$ is solved for X .

If $\text{trans} = \text{'C'}$, then $(\text{sub}(A))^H * X = \text{sub}(B)$ is solved for X .

n (global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).

bwl (global) INTEGER. The number of subdiagonals within the band of A ($0 \leq bwl \leq n-1$).

bwu (global) INTEGER. The number of superdiagonals within the band of A ($0 \leq bwu \leq n-1$).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($nrhs \geq 0$).

a, b (local)

REAL for psdbtrs

DOUBLE PRECISON for pddbtrs

COMPLEX for pcdptrs

DOUBLE COMPLEX for pzdbtrs.

Pointers into the local memory to arrays of local dimension $a(\text{lld}_a, \text{LOCc}(ja+n-1))$ and $b(\text{lld}_b, \text{LOCc}(nrhs))$, respectively.

On entry, the array *a* contains details of the LU factorization of the band matrix A , as computed by p?dbtrf.

On entry, the array *b* contains the local pieces of the right hand side distributed matrix $\text{sub}(B)$.

ja (global) INTEGER. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A .

If $\text{desca}(\text{dtype}_-) = 501$, then $\text{dlen}_- \geq 7$;

else if $\text{desca}(\text{dtype}_-) = 1$, then $\text{dlen}_- \geq 9$.

ib (global) INTEGER. The row index in the global array B that points to the first row of the matrix to be operated on (which may be either all of B or a submatrix of B).

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix B .

If $\text{descb}(\text{dtype}_-) = 502$, then $\text{dlen}_- \geq 7$;

else if $\text{descb}(\text{dtype}_-) = 1$, then $\text{dlen}_- \geq 9$.

af, work (local)

REAL for psdbtrs
 DOUBLE PRECISION for pdabtrs
 COMPLEX for pcdbtrs
 DOUBLE COMPLEX for pzdbtrs.

Arrays of dimension (*laf*) and (*lwork*), respectively. The array *af* contains auxiliary Fillin space. Fillin is created during the factorization routine p?dbtrf and this is stored in *af*.

The array *work* is a workspace array.

laf

(local) INTEGER. The dimension of the array *af*.

Must be $laf \geq NB * (bw1 + bwu) + 6 * (\max(bw1, bwu))^2$.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

desca

(local or global) INTEGER. The size of the array *work*, must be at least

$lwork \geq (\max(bw1, bwu))^2$.

Output Parameters

b

On exit, this array contains the local pieces of the solution distributed matrix *X*.

work(1)

On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

INTEGER. If *info*=0, the execution is successful. *info* < 0:

if the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?dttrs

Solves a system of linear equations with a diagonally dominant-like tridiagonal distributed matrix using the factorization computed by p?dttrf.

Syntax

Fortran:

```
call psdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work,
lwork, info)

call pdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work,
lwork, info)

call pcdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work,
lwork, info)

call pzdttrs(trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

C:

```

void psdttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , float *dl , float *d ,
              float *du , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib ,
              MKL_INT *descb , float *af , MKL_INT *laf , float *work ,
              MKL_INT *lwork , MKL_INT *info );

void pddttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , double *dl , double *d ,
               double *du , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
               MKL_INT *descb , double *af , MKL_INT *laf , double *work ,
               MKL_INT *lwork , MKL_INT *info );

void pcdttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *dl ,
               MKL_Complex8 *d , MKL_Complex8 *du , MKL_INT *ja , MKL_INT *desca ,
               MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb , MKL_Complex8 *af ,
               MKL_INT *laf , MKL_Complex8 *work , MKL_INT *lwork ,
               MKL_INT *info );

void pzdttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *dl ,
               MKL_Complex16 *d , MKL_Complex16 *du , MKL_INT *ja , MKL_INT *desca ,
               MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb , MKL_Complex16 *af ,
               MKL_INT *laf , MKL_Complex16 *work , MKL_INT *lwork ,
               MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The p?dttrs routine solves for X one of the systems of equations:

```

sub(A)*X = sub(B),
(sub(A))^T*X = sub(B), or
(sub(A))^H*X = sub(B),

```

where $\text{sub}(A) = (1:n, ja:ja+n-1)$; is a diagonally dominant-like tridiagonal distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses the LU factorization computed by p?dttrf.

Input Parameters

<i>trans</i>	(global) CHARACTER*1. Must be 'N' or 'T' or 'C'.
	Indicates the form of the equations:
	If <i>trans</i> = 'N', then $\text{sub}(A)*X = \text{sub}(B)$ is solved for X .
	If <i>trans</i> = 'T', then $(\text{sub}(A))^T*X = \text{sub}(B)$ is solved for X .
	If <i>trans</i> = 'C', then $(\text{sub}(A))^H*X = \text{sub}(B)$ is solved for X .
<i>n</i>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($nrhs \geq 0$).
<i>dl, d, du</i>	(local)
	REAL for psdttrs
	DOUBLE PRECISION for pddttrs
	COMPLEX for pcdttrs
	DOUBLE COMPLEX for pzdttrs.
	Pointers to the local arrays of dimension (<i>desca(nb_)</i>) each.

On entry, these arrays contain details of the factorization. Globally, $d(1)$ and $du(n)$ are not referenced; dl and du must be aligned with d .

ja (global) INTEGER. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).

desca (global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A . If $desca(dtype_)$ = 501 or 502, then $dlen_ \geq 7$;
else if $desca(dtype_)$ = 1, then $dlen_ \geq 9$.

b (local) Same type as d .
 Pointer into the local memory to an array of local dimension
 $b(lld_b, LOCc(nrhs))$.
 On entry, the array b contains the local pieces of the n -by- $nrhs$ right hand side distributed matrix sub(B).

ib (global) INTEGER. The row index in the global array B that points to the first row of the matrix to be operated on (which may be either all of B or a submatrix of B).

descb (global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix B .
If $descb(dtype_)$ = 502, then $dlen_ \geq 7$;
else if $descb(dtype_)$ = 1, then $dlen_ \geq 9$.

af, work (local) REAL for psdttrs
 DOUBLE PRECISION for pddttrs
 COMPLEX for pcdttrs
 DOUBLE COMPLEX for pzdttrs.

Arrays of dimension (*laf*) and (*lwork*), respectively.

The array *af* contains auxiliary Fillin space. Fillin is created during the factorization routine $p?dttrf$ and this is stored in *af*. If a linear system is to be solved using $p?dttrs$ after the factorization routine, *af* must not be altered.

The array *work* is a workspace array.

laf (local) INTEGER. The dimension of the array *af*.
 Must be $laf \geq NB * (bwl+bwu) + 6 * (bwl+bwu) * (bwl+2 * bwu)$.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

lwork (local or global) INTEGER. The size of the array *work*, must be at least
 $lwork \geq 10 * NPCOL + 4 * nrhs$.

Output Parameters

<i>b</i>	On exit, this array contains the local pieces of the solution distributed matrix X .
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of $\lceil \text{work} \rceil$ required for optimum performance.
<i>info</i>	INTEGER. If $\text{info}=0$, the execution is successful. $\text{info} < 0$: if the i -th argument is an array and the j -th entry had an illegal value, then $\text{info} = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.

p?potrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian distributed positive-definite matrix.

Syntax

Fortran:

```
call pspotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pdpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pcpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pzpotrs(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

C:

```
void pspotrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
MKL_INT *info );
void pdpotrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
void pcpotrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
void pzpotrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?potrs routine solves for X a system of distributed linear equations in the form:

$\text{sub}(A) * X = \text{sub}(B)$,

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ is an n -by- n real symmetric or complex Hermitian positive definite distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, jb:jb+nrhs-1)$.

This routine uses Cholesky factorization

`sub(A) = UH*U, or sub(A) = L*LH`

computed by `p?potrf`.

Input Parameters

<code>uplo</code>	(global) CHARACTER*1. Must be 'U' or 'L'. If <code>uplo</code> = 'U', upper triangle of <code>sub(A)</code> is stored; If <code>uplo</code> = 'L', lower triangle of <code>sub(A)</code> is stored.
<code>n</code>	(global) INTEGER. The order of the distributed submatrix <code>sub(A)</code> ($n \geq 0$).
<code>nrhs</code>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix <code>sub(B)</code> ($nrhs \geq 0$).
<code>a, b</code>	(local) REAL for <code>pspotrs</code> DOUBLE PRECISION for <code>pdpotrs</code> COMPLEX for <code>pcpotrs</code> DOUBLE COMPLEX for <code>pzpotrs</code> . Pointers into the local memory to arrays of local dimension <code>a(lld_a, LOCc(ja+n-1))</code> and <code>b(lld_b, LOCc(jb+nrhs-1))</code> , respectively. The array <code>a</code> contains the factors <code>L</code> or <code>U</code> from the Cholesky factorization <code>sub(A) = L*L^H</code> or <code>sub(A) = U^H*U</code> , as computed by <code>p?potrf</code> . On entry, the array <code>b</code> contains the local pieces of the right hand sides <code>sub(B)</code> .
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array <code>A</code> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<code>desca</code>	(global and local) INTEGER array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix <code>A</code> .
<code>ib, jb</code>	(global) INTEGER. The row and column indices in the global array <code>B</code> indicating the first row and the first column of the submatrix <code>sub(B)</code> , respectively.
<code>descb</code>	(local) INTEGER array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix <code>B</code> .

Output Parameters

<code>b</code>	Overwritten by the local pieces of the solution matrix <code>X</code> .
<code>info</code>	INTEGER. If <code>info=0</code> , the execution is successful. <code>info < 0</code> : if the i -th argument is an array and the j -th entry had an illegal value, then <code>info = -(i*100+j)</code> ; if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

p?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian positive-definite band matrix.

Syntax

Fortran:

```
call pspbtrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork,
info)
```

```
call pdpbtrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork,
info)
```

```
call pcpbtrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork,
info)
```

```
call pzpbtrs(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork,
info)
```

C:

```
void pspbtrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , float *a ,
MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *af ,
MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pdpbtrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , double *a ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double *af ,
MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcpbtrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , MKL_Complex8 *a ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzpbtrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , MKL_Complex16
*a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?pbtrs routine solves for X a system of distributed linear equations in the form:

$\text{sub}(A) * X = \text{sub}(B)$,

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an n -by- n real symmetric or complex Hermitian positive definite distributed band matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This routine uses Cholesky factorization

$\text{sub}(A) = P * U^H * U * P^T$, or $\text{sub}(A) = P * L * L^H * P^T$

computed by p?pbtrf.

Input Parameters

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

If $uplo = 'U'$, upper triangle of $\text{sub}(A)$ is stored;
 If $uplo = 'L'$, lower triangle of $\text{sub}(A)$ is stored.

n (global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).

bw (global) INTEGER. The number of superdiagonals of the distributed matrix if $uplo = 'U'$, or the number of subdiagonals if $uplo = 'L'$ ($bw \geq 0$).

nrhs (global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($nrhs \geq 0$).

a, b (local)

REAL for pspbtrs
 DOUBLE PRECISION for pdpbtrs
 COMPLEX for pcpbtrs
 DOUBLE COMPLEX for pzpbtrs.

Pointers into the local memory to arrays of local dimension $a(lld_a, LOCc(ja+n-1))$ and $b(lld_b, LOCc(nrhs-1))$, respectively.

The array *a* contains the permuted triangular factor *U* or *L* from the Cholesky factorization $\text{sub}(A) = P^* U^{H*} U^* P^T$, or $\text{sub}(A) = P^* L^* L^{H*} P^T$ of the band matrix *A*, as returned by [p?pbtrf](#).

On entry, the array *b* contains the local pieces of the *n*-by-*nrhs* right hand side distributed matrix $\text{sub}(B)$.

ja (global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

If $\text{desca}(\text{dtype}_) = 501$, then $dlen_ \geq 7$;
 else if $\text{desca}(\text{dtype}_) = 1$, then $dlen_ \geq 9$.

ib (global) INTEGER. The row index in the global array *B* indicating the first row of the submatrix $\text{sub}(B)$.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

If $\text{descb}(\text{dtype}_) = 502$, then $dlen_ \geq 7$;
 else if $\text{descb}(\text{dtype}_) = 1$, then $dlen_ \geq 9$.

af, work (local) Arrays, same type as *a*.

The array *af* is of dimension (*laf*). It contains auxiliary Fillin space. Fillin is created during the factorization routine [p?dbtrf](#) and this is stored in *af*.

The array *work* is a workspace array of dimension *lwork*.

laf (local) INTEGER. The dimension of the array *af*.

Must be $\text{laf} \geq nrhs * bw$.

If laf is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af(1)$.

$lwork$ (local or global) INTEGER. The size of the array $work$, must be at least $lwork \geq bw^2$.

Output Parameters

b On exit, if $info=0$, this array contains the local pieces of the n -by- $nrhs$ solution distributed matrix X .

$work(1)$ On exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance.

$info$ INTEGER. If $info=0$, the execution is successful.

$info < 0$:

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?pttrs

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal distributed matrix using the factorization computed by p?pttrf.

Syntax

Fortran:

```
call pspttrs(n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pdpttrs(n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pcpttrs(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
call pzpttrs(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

C:

```
void pspttrs (MKL_INT *n , MKL_INT *nrhs , float *d , float *e , MKL_INT *ja , MKL_INT
*desca , float *b , MKL_INT *ib , MKL_INT *descb , float *af , MKL_INT *laf , float
*work , MKL_INT *lwork , MKL_INT *info );
void pdpttrs (MKL_INT *n , MKL_INT *nrhs , double *d , double *e , MKL_INT *ja ,
MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double *af , MKL_INT
*laf , double *work , MKL_INT *lwork , MKL_INT *info );
void pcpttrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *d , MKL_Complex8 *e ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );
void pzpttrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *d , MKL_Complex16 *e ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?pttrs` routine solves for X a system of distributed linear equations in the form:

`sub(A) * X = sub(B)`,

where `sub(A) = A(1:n, ja:ja+n-1)` is an n -by- n real symmetric or complex Hermitian positive definite tridiagonal distributed matrix, and `sub(B)` denotes the distributed matrix `B(ib:ib+n-1, 1:nrhs)`.

This routine uses the factorization

`sub(A) = P * L * D * LH * PT`, or `sub(A) = P * UH * D * U * PT`

computed by `p?pttrf`.

Input Parameters

<code>uplo</code>	(global, used in complex flavors only) CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '. If <code>uplo</code> = ' <code>U</code> ', upper triangle of <code>sub(A)</code> is stored; If <code>uplo</code> = ' <code>L</code> ', lower triangle of <code>sub(A)</code> is stored.
<code>n</code>	(global) INTEGER. The order of the distributed submatrix <code>sub(A)</code> ($n \geq 0$).
<code>nrhs</code>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix <code>sub(B)</code> ($nrhs \geq 0$).
<code>d, e</code>	(local) REAL for <code>pspttrs</code> DOUBLE PRECISION for <code>pdpttrs</code> COMPLEX for <code>pcpttrs</code> DOUBLE COMPLEX for <code>pzpttrs</code> . Pointers into the local memory to arrays of dimension (<code>desca(nb_)</code>) each. These arrays contain details of the factorization as returned by <code>p?pttrf</code>
<code>ja</code>	(global) INTEGER. The index in the global array <code>A</code> that points to the start of the matrix to be operated on (which may be either all of <code>A</code> or a submatrix of <code>A</code>).
<code>desca</code>	(global and local) INTEGER array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix <code>A</code> . If <code>desca(dtype_) = 501</code> or <code>502</code> , then <code>dlen_ ≥ 7</code> ; else if <code>desca(dtype_) = 1</code> , then <code>dlen_ ≥ 9</code> .
<code>b</code>	(local) Same type as <code>d, e</code> . Pointer into the local memory to an array of local dimension <code>b(lld_b, LOCc(nrhs))</code> .

On entry, the array b contains the local pieces of the n -by- $nrhs$ right hand side distributed matrix sub(B).

ib	(global) INTEGER. The row index in the global array B that points to the first row of the matrix to be operated on (which may be either all of B or a submatrix of B).
$descb$	(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix B . If $descb(dtype_)$ = 502, then $dlen_ \geq 7$; else if $descb(dtype_)$ = 1, then $dlen_ \geq 9$.
$af, work$	(local) REAL for pspttrs DOUBLE PRECISION for pdpttrs COMPLEX for pcpttrs DOUBLE COMPLEX for pzpttrs. Arrays of dimension (laf) and ($lwork$), respectively. The array af contains auxiliary Fillin space. Fillin is created during the factorization routine p?pttrf and this is stored in af . The array $work$ is a workspace array.
laf	(local) INTEGER. The dimension of the array af . Must be $laf \geq NB+2$. If laf is not large enough, an error code is returned and the minimum acceptable size will be returned in $af(1)$.
$lwork$	(local or global) INTEGER. The size of the array $work$, must be at least $lwork \geq (10+2*\min(100,nrhs))*NPCOL+4*nrhs$.

Output Parameters

b	On exit, this array contains the local pieces of the solution distributed matrix X .
$work(1)$	On exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	INTEGER. If $info=0$, the execution is successful. $info < 0$: if the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?trtrs

Solves a system of linear equations with a triangular distributed matrix.

Syntax

Fortran:

```
call pstrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pdtrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pctrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pztrtrs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

C:

```
void pstrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
void pdtrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
void pctrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
void pztrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?trtrs routine solves for X one of the following systems of linear equations:

```
sub(A)*X = sub(B),
(sub(A))T*X = sub(B), or
(sub(A))H*X = sub(B),
```

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ is a triangular distributed matrix of order n , and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, jb:jb+nrhs-1)$.

A check is made to verify that $\text{sub}(A)$ is nonsingular.

Input Parameters

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

Indicates whether $\text{sub}(A)$ is upper or lower triangular:

If *uplo* = 'U', then $\text{sub}(A)$ is upper triangular.

If *uplo* = 'L', then $\text{sub}(A)$ is lower triangular.

trans (global) CHARACTER*1. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', then $\text{sub}(A)*X = \text{sub}(B)$ is solved for X .

If *trans* = 'T', then $\text{sub}(A)^T*X = \text{sub}(B)$ is solved for X .

If $\text{trans} = \text{'C'}$, then $\text{sub}(A)^H X = \text{sub}(B)$ is solved for X .

diag

(global) CHARACTER*1. Must be ' N ' or ' U '.

If $\text{diag} = \text{'N'}$, then $\text{sub}(A)$ is not a unit triangular matrix.

If $\text{diag} = \text{'U'}$, then $\text{sub}(A)$ is unit triangular.

n

(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).

nrhs

(global) INTEGER. The number of right-hand sides; i.e., the number of columns of the distributed matrix $\text{sub}(B)$ ($\text{nrhs} \geq 0$).

a, b

(local)

REAL for pstrtrs

DOUBLE PRECISION for pdtrtrs

COMPLEX for pcptrtrs

DOUBLE COMPLEX for pzptrtrs.

Pointers into the local memory to arrays of local dimension

$a(\text{lld}_a, \text{LOCc}(ja+n-1))$ and $b(\text{lld}_b, \text{LOCc}(jb+nrhs-1))$, respectively.

The array a contains the local pieces of the distributed triangular matrix $\text{sub}(A)$.

If $\text{uplo} = \text{'U'}$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced.

If $\text{uplo} = \text{'L'}$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

If $\text{diag} = \text{'U'}$, the diagonal elements of $\text{sub}(A)$ are also not referenced and are assumed to be 1.

On entry, the array b contains the local pieces of the right hand side distributed matrix $\text{sub}(B)$.

ia, ja

(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desc_a

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .

ib, jb

(global) INTEGER. The row and column indices in the global array B indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.

desc_b

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix B .

Output Parameters

b

On exit, if $\text{info}=0$, $\text{sub}(B)$ is overwritten by the solution matrix X .

info INTEGER. If *info*=0, the execution is successful.

info < 0:

if the *i*th argument is an array and the *j*th entry had an illegal value, then
info = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal
value, then *info* = -*i*;

info > 0:

if *info* = *i*, the *i*-th diagonal element of sub(*A*) is zero, indicating that the
submatrix is singular and the solutions *X* have not been computed.

Routines for Estimating the Condition Number

This section describes the ScaLAPACK routines for estimating the condition number of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations. Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the *reciprocal* condition number.

p?gecon

Estimates the reciprocal of the condition number of a general distributed matrix in either the 1-norm or the infinity-norm.

Syntax

Fortran:

```
call psgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork,
info)

call pdgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork,
info)

call pcgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork,
info)

call pzgecon(norm, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork,
info)
```

C:

```
void psgecon (char *norm , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *anorm , float *rcond , float *work , MKL_INT *lwork , MKL_INT *iwork ,
MKL_INT *liwork , MKL_INT *info );

void pdgecon (char *norm , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *anorm , double *rcond , double *work , MKL_INT *lwork ,
MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pcgecon (char *norm , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *anorm , float *rcond , MKL_Complex8 *work , MKL_INT *lwork ,
float *rwork , MKL_INT *lrwork , MKL_INT *info );

void pzgecon (char *norm , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *anorm , double *rcond , MKL_Complex16 *work , MKL_INT
*lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?gecon` routine estimates the reciprocal of the condition number of a general distributed real/complex matrix $\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+n-1)$ in either the 1-norm or infinity-norm, using the *LU* factorization computed by `p?getrf`.

An estimate is obtained for $\|(\text{sub}(A))^{-1}\|$, and the reciprocal of the condition number is computed as

... : ! : ! : ! : ! : ! : ! : ! : ! : ! : ! : ! : ! : ! : !

Input Parameters

norm

(global) CHARACTER*1. Must be '1' or 'O' or 'I'.

Specifies whether the 1-norm condition number or the infinity-norm condition number is required.

If *norm* = '1' or 'O', then the 1-norm is used;

If *norm* = 'I', then the infinity-norm is used.

n

(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).

a

(local)

REAL for psgecon

DOUBLE PRECISION for pdgecon

COMPLEX for pcgecon

DOUBLE COMPLEX for pzgecon.

Pointer into the local memory to an array of dimension $a(\text{lld}_a, \text{LOCc}(\text{ja} + n - 1))$.

The array *a* contains the local pieces of the factors *L* and *U* from the factorization $\text{sub}(A) = P*L*U$; the unit diagonal elements of *L* are not stored.

ia, ja

(global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

anorm

(global) REAL for single precision flavors, DOUBLE PRECISION for double precision flavors.

If *norm* = '1' or 'O', the 1-norm of the original distributed matrix $\text{sub}(A)$;

If *norm* = 'I', the infinity-norm of the original distributed matrix $\text{sub}(A)$.

<i>work</i>	(local) REAL for psgecon DOUBLE PRECISION for pdgecon COMPLEX for pcgecon DOUBLE COMPLEX for pzgecon.
	The array <i>work</i> of dimension (<i>lwork</i>) is a workspace array.
<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> . <i>For real flavors:</i> <i>lwork</i> must be at least $lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a)) + 2 * LOCc(n + \text{mod}(ja-1, nb_a)) + \max(2, \max(nb_a * \max(1, \text{ceil}(NPROW-1, NPCOL)), LOCc(n + \text{mod}(ja-1, nb_a)) + nb_a * \max(1, \text{ceil}(NPCOL-1, NPROW)))) .$
	<i>For complex flavors:</i> <i>lwork</i> must be at least $lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a)) + \max(2, \max(nb_a * \text{ceil}(NPROW-1, NPCOL), LOCc(n + \text{mod}(ja-1, nb_a)) + nb_a * \text{ceil}(NPCOL-1, NPROW))) .$
	<i>LOCr</i> and <i>LOCc</i> values can be computed using the ScaLAPACK tool function <i>numroc</i> ; <i>NPROW</i> and <i>NPCOL</i> can be determined by calling the subroutine <i>blacs_gridinfo</i> .
<i>iwork</i>	(local) INTEGER. Workspace array, size (<i>liwork</i>). Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The dimension of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ia-1, mb_a)) .$
<i>rwork</i>	(local) REAL for pcgecon DOUBLE PRECISION for pzgecon Workspace array, size (<i>lrwork</i>). Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The dimension of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq \max(1, 2 * LOCc(n + \text{mod}(ja-1, nb_a))) .$

Output Parameters

<i>rcond</i>	(global) REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The reciprocal of the condition number of the distributed matrix <i>sub(A)</i> . See Description.
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.

<i>iwork</i> (1)	On exit, <i>iwork</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance (for real flavors).
<i>rwork</i> (1)	On exit, <i>rwork</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful.
	<i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>) ; if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?pocon

Estimates the reciprocal of the condition number (in the 1 - norm) of a symmetric / Hermitian positive-definite distributed matrix.

Syntax

Fortran:

```
call pspocon(uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork,
info)

call pdpocon(uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, iwork, liwork,
info)

call pcpocon(uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork,
info)

call pzpocon(uplo, n, a, ia, ja, desca, anorm, rcond, work, lwork, rwork, lrwork,
info)
```

C:

```
void pspocon (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *anorm , float *rcond , float *work , MKL_INT *lwork , MKL_INT *iwork ,
MKL_INT *liwork , MKL_INT *info );

void pdpocon (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *anorm , double *rcond , double *work , MKL_INT *lwork ,
MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pcpocon (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *anorm , float *rcond , MKL_Complex8 *work , MKL_INT *lwork ,
float *rwork , MKL_INT *lrwork , MKL_INT *info );

void pzpocon (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *anorm , double *rcond , MKL_Complex16 *work , MKL_INT
*lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?pocon routine estimates the reciprocal of the condition number (in the 1 - norm) of a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$, using the Cholesky factorization $\text{sub}(A) = U^H \cdot U$ or $\text{sub}(A) = L \cdot L^H$ computed by p?potrf.

An estimate is obtained for $\|(\text{sub}(A))^{-1}\|$, and the reciprocal of the condition number is computed as

www.ijerph.org | ISSN: 1660-4601 | DOI: 10.3390/ijerph17030894

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the factor stored in sub(<i>A</i>) is upper or lower triangular. If <i>uplo</i> = 'U', sub(<i>A</i>) stores the upper triangular factor <i>U</i> of the Cholesky factorization sub (<i>A</i>) = $U^H * U$. If <i>uplo</i> = 'L', sub(<i>A</i>) stores the lower triangular factor <i>L</i> of the Cholesky factorization sub (<i>A</i>) = $L * L^H$.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix sub(<i>A</i>) ($n \geq 0$).
<i>a</i>	(local) REAL for pspocon DOUBLE PRECISION for pdpocon COMPLEX for pcpocon DOUBLE COMPLEX for pzpocon. Pointer into the local memory to an array of dimension <i>a</i> (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). The array <i>a</i> contains the local pieces of the factors <i>L</i> or <i>U</i> from the Cholesky factorization sub (<i>A</i>) = $U^H * U$, or sub (<i>A</i>) = $L * L^H$, as computed by p?potrf.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix sub(<i>A</i>), respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>anorm</i>	(global) REAL for single precision flavors, DOUBLE PRECISION for double precision flavors. The 1-norm of the symmetric/Hermitian distributed matrix sub(<i>A</i>).
<i>work</i>	(local) REAL for pspocon DOUBLE PRECISION for pdpocon COMPLEX for pcpocon DOUBLE COMPLEX for pzpocon. The array <i>work</i> of dimension (<i>lwork</i>) is a workspace array.

<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> .
	<i>For real flavors:</i>
	<i>lwork</i> must be at least
	$lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a)) + 2 * LOCc(n + \text{mod}(ja-1, nb_a)) + \max(2, \max(nb_a * \text{ceil}(NPROW-1, NPCOL), LOCc(n + \text{mod}(ja-1, nb_a)) + nb_a * \text{ceil}(NPCOL-1, NPROW))).$
	<i>For complex flavors:</i>
	<i>lwork</i> must be at least
	$lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a)) + \max(2, \max(nb_a * \max(1, \text{ceil}(NPROW-1, NPCOL)), LOCc(n + \text{mod}(ja-1, nb_a)) + nb_a * \max(1, \text{ceil}(NPCOL-1, NPROW)))).$
<i>iwork</i>	(local) INTEGER. Workspace array, size (<i>liwork</i>). Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The dimension of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ia-1, mb_a))$.
<i>rwork</i>	(local) REAL for pcpocon DOUBLE PRECISION for pzpocon Workspace array, size (<i>lrwork</i>). Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The dimension of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq 2 * LOCc(n + \text{mod}(ja-1, nb_a))$.

Output Parameters

<i>rcond</i>	(global) REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. The reciprocal of the condition number of the distributed matrix $\text{sub}(A)$.
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork(1)</i>	On exit, <i>iwork(1)</i> contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork(1)</i>	On exit, <i>rwork(1)</i> contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> th argument is an array and the <i>j</i> th entry had an illegal value, then <i>info</i> = $-(i * 100 + j)$; if the <i>i</i> th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

p?trcon

Estimates the reciprocal of the condition number of a triangular distributed matrix in either 1-norm or infinity-norm.

Syntax

Fortran:

```
call pstrcon(
  norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork, iwork, liwork, info)
call pdtrcon(
  norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork, iwork, liwork, info)
call pcptrcon(
  norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork, rwork, lrwork, info)
call pzptrcon(
  norm, uplo, diag, n, a, ia, ja, desca, rcond, work, lwork, rwork, lrwork, info)
```

C:

```
void pstrcon (char *norm , char *uplo , char *diag , MKL_INT *n , float *a , MKL_INT
  *ia , MKL_INT *ja , MKL_INT *desca , float *rcond , float *work , MKL_INT *lwork ,
  MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
void pdtrcon (char *norm , char *uplo , char *diag , MKL_INT *n , double *a , MKL_INT
  *ia , MKL_INT *ja , MKL_INT *desca , double *rcond , double *work , MKL_INT *lwork ,
  MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
void pcptrcon (char *norm , char *uplo , char *diag , MKL_INT *n , MKL_Complex8 *a ,
  MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *rcond , MKL_Complex8 *work ,
  MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *info );
void pzptrcon (char *norm , char *uplo , char *diag , MKL_INT *n , MKL_Complex16 *a ,
  MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *rcond , MKL_Complex16 *work ,
  MKL_INT *lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?trcon routine estimates the reciprocal of the condition number of a triangular distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$, in either the 1-norm or the infinity-norm.

The norm of $\text{sub}(A)$ is computed and an estimate is obtained for $\|(\text{sub}(A))^{-1}\|$, then the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|\text{sub}(A)\| \times \|(\text{sub}(A))^{-1}\|}$$

Input Parameters

<i>norm</i>	(global) CHARACTER*1. Must be '1' or 'O' or 'I'.
	Specifies whether the 1-norm condition number or the infinity-norm condition number is required.

If $norm = '1'$ or ' ∞ ', then the 1-norm is used;
 If $norm = 'I'$, then the infinity-norm is used.

uplo
 (global) CHARACTER*1. Must be ' U ' or ' L '.
 If $uplo = 'U'$, $\text{sub}(A)$ is upper triangular. If $uplo = 'L'$, $\text{sub}(A)$ is lower triangular.

diag
 (global) CHARACTER*1. Must be ' N ' or ' U '.
 If $diag = 'N'$, $\text{sub}(A)$ is non-unit triangular. If $diag = 'U'$, $\text{sub}(A)$ is unit triangular.

n
 (global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$, ($n \geq 0$).

a
 (local)
 REAL for pstrcon
 DOUBLE PRECISION for pdtrcon
 COMPLEX for pcstrcon
 DOUBLE COMPLEX for pztrcon.
 Pointer into the local memory to an array of dimension
 $a(1:ld_a, 1:LOCc(ja+n-1))$.
 The array *a* contains the local pieces of the triangular distributed matrix $\text{sub}(A)$.
 If $uplo = 'U'$, the leading n -by- n upper triangular part of this distributed matrix contains the upper triangular matrix, and its strictly lower triangular part is not referenced.
 If $uplo = 'L'$, the leading n -by- n lower triangular part of this distributed matrix contains the lower triangular matrix, and its strictly upper triangular part is not referenced.
 If $diag = 'U'$, the diagonal elements of $\text{sub}(A)$ are also not referenced and are assumed to be 1.

ia, ja
 (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca
 (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work
 (local)
 REAL for pstrcon
 DOUBLE PRECISION for pdtrcon
 COMPLEX for pcstrcon
 DOUBLE COMPLEX for pztrcon.
 The array *work* of dimension (*lwork*) is a workspace array.

lwork
 (local or global) INTEGER. The dimension of the array *work*.

For real flavors:

lwork must be at least

$$\begin{aligned} lwork \geq 2 * LOCr(n+mod(ia-1, mb_a)) + LOCc(n+mod(ja-1, nb_a)) + \\ \max(2, \max(nb_a * \max(1, \text{ceil}(NPROW-1, NPCOL)), \\ LOCc(n+mod(ja-1, nb_a)) + nb_a * \max(1, \text{ceil}(NPCOL-1, NPROW)))) . \end{aligned}$$

For complex flavors:

lwork must be at least

$$\begin{aligned} lwork \geq 2 * LOCr(n+mod(ia-1, mb_a)) + \max(2, \max(nb_a * \text{ceil}(\\ NPROW-1, NPCOL), \\ LOCc(n+mod(ja-1, nb_a)) + nb_a * \text{ceil}(NPCOL-1, NPROW)))) . \end{aligned}$$

iwork

(local) INTEGER. Workspace array, size (*lwork*). Used in real flavors only.

liwork

(local or global) INTEGER. The dimension of the array *iwork*; used in real flavors only. Must be at least

$$liwork \geq LOCr(n+mod(ia-1, mb_a)) .$$

rwork

(local) REAL for pcpoco

DOUBLE PRECISION for pzpocon

Workspace array, size (*lrwork*). Used in complex flavors only.

lrwork

(local or global) INTEGER. The dimension of the array *rwork*; used in complex flavors only. Must be at least

$$lrwork \geq LOCc(n+mod(ja-1, nb_a)) .$$

Output Parameters

rcond

(global) REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

The reciprocal of the condition number of the distributed matrix sub(*A*).

work(1)

On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1)

On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1)

On exit, *rwork*(1) contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info

(global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*) ; if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Refining the Solution and Estimating Its Error

This section describes the ScalAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see [Routines for Matrix Factorization](#) and [Solving Systems of Linear Equations](#)).

p?gerfs

Improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

Syntax

Fortran:

```
call psgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)

call pdgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)

call pcgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)

call pzgerfs(trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, ipiv, b, ib, jb,
descb, x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

C:

```
void psgerfs (char *trans , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT
*descaf , MKL_INT *ipiv , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *ferr , float *berr ,
float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pdgerfs (char *trans , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT
*descaf , MKL_INT *ipiv , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *ferr , double *berr ,
double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pcgerfs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *iaf , MKL_INT *jaf ,
MKL_INT *descaf , MKL_INT *ipiv , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float
*ferr , float *berr , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT
*lrwork , MKL_INT *info );

void pzgerfs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *iaf , MKL_INT *jaf ,
MKL_INT *descaf , MKL_INT *ipiv , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
double *ferr , double *berr , MKL_Complex16 *work , MKL_INT *lwork , double *rwork ,
MKL_INT *lrwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?gerfs` routine improves the computed solution to one of the systems of linear equations

`sub(A) *sub(X) = sub(B)`,

`sub(A)T*sub(X) = sub(B)`, or

`sub(A)H*sub(X) = sub(B)` and provides error bounds and backward error estimates for the solution.

Here `sub(A) = A(ia:ia+n-1, ja:ja+n-1)`, `sub(B) = B(ib:ib+n-1, jb:jb+nrhs-1)`, and `sub(X) = X(ix:ix+n-1, jx:jx+nrhs-1)`.

Input Parameters

<code>trans</code>	(global) CHARACTER*1. Must be ' <code>N</code> ' or ' <code>T</code> ' or ' <code>C</code> '.
	Specifies the form of the system of equations:
	If <code>trans = 'N'</code> , the system has the form <code>sub(A) *sub(X) = sub(B)</code> (No transpose);
	If <code>trans = 'T'</code> , the system has the form <code>sub(A)^T*sub(X) = sub(B)</code> (Transpose);
	If <code>trans = 'C'</code> , the system has the form <code>sub(A)^H*sub(X) = sub(B)</code> (Conjugate transpose).
<code>n</code>	(global) INTEGER. The order of the distributed submatrix <code>sub(A)</code> ($n \geq 0$).
<code>nrhs</code>	(global) INTEGER. The number of right-hand sides, i.e., the number of columns of the matrices <code>sub(B)</code> and <code>sub(X)</code> ($nrhs \geq 0$).
<code>a, af, b, x</code>	(local) REAL for <code>psgerfs</code> DOUBLE PRECISION for <code>pdgerfs</code> COMPLEX for <code>pcgerfs</code> DOUBLE COMPLEX for <code>pzgerfs</code> . Pointers into the local memory to arrays of local dimension <code>a(lld_a, LOCc(ja+n-1)), af(lld_af, LOCc(jaf+n-1)), b(lld_b, LOCc(jb+nrhs-1))</code> , and <code>x(lld_x, LOCc(jx+nrhs-1))</code> , respectively. The array <code>a</code> contains the local pieces of the distributed matrix <code>sub(A)</code> . The array <code>af</code> contains the local pieces of the distributed factors of the matrix <code>sub(A) = P*L*U</code> as computed by <code>p?getrf</code> . The array <code>b</code> contains the local pieces of the distributed matrix of right hand sides <code>sub(B)</code> . On entry, the array <code>x</code> contains the local pieces of the distributed solution matrix <code>sub(X)</code> .

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>iaf, jaf</i>	(global) INTEGER. The row and column indices in the global array <i>AF</i> indicating the first row and the first column of the submatrix $\text{sub}(AF)$, respectively.
<i>descaf</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>AF</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the global array <i>X</i> indicating the first row and the first column of the submatrix $\text{sub}(X)$, respectively.
<i>descx</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>X</i> .
<i>ipiv</i>	(local) INTEGER. Array, dimension $\text{LOCr}(m_af + mb_af)$. This array contains pivoting information as computed by p?getrf . If $ipiv(i)=j$, then the local row <i>i</i> was swapped with the global row <i>j</i> . This array is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for <i>psgerfs</i> DOUBLE PRECISION for <i>pdgerfs</i> COMPLEX for <i>pcgerfs</i> DOUBLE COMPLEX for <i>pzgerfs</i> . The array <i>work</i> of dimension (<i>lwork</i>) is a workspace array.
<i>lwork</i>	(local or global) INTEGER. The dimension of the array <i>work</i> . <i>For real flavors:</i> <i>lwork</i> must be at least $lwork \geq 3 * \text{LOCr}(n + \text{mod}(ia-1, mb_a))$ <i>For complex flavors:</i> <i>lwork</i> must be at least $lwork \geq 2 * \text{LOCr}(n + \text{mod}(ia-1, mb_a))$

<i>iwork</i>	(local) INTEGER. Workspace array, size (<i>liwork</i>). Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The dimension of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOC_r(n + \text{mod}(ib-1, mb_b))$.
<i>rwork</i>	(local) REAL for pcgerfs DOUBLE PRECISION for pzgerfs Workspace array, size (<i>lrwork</i>). Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The dimension of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq LOC_r(n + \text{mod}(ib-1, mb_b))$.

Output Parameters

<i>x</i>	On exit, contains the improved solution vectors.
<i>ferr, berr</i>	REAL for single precision flavors. DOUBLE PRECISION for double precision flavors. Arrays, dimension $LOC_c(jb + nrhs - 1)$ each. The array <i>ferr</i> contains the estimated forward error bound for each solution vector of $\text{sub}(X)$. If XTRUE is the true solution corresponding to $\text{sub}(X)$, <i>ferr</i> is an estimated upper bound for the magnitude of the largest element in $(\text{sub}(X) - XTRUE)$ divided by the magnitude of the largest element in $\text{sub}(X)$. The estimate is as reliable as the estimate for <i>rcond</i> , and is almost always a slight overestimate of the true error. This array is tied to the distributed matrix <i>X</i> . The array <i>berr</i> contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of $\text{sub}(A)$ or $\text{sub}(B)$ that makes $\text{sub}(X)$ an exact solution). This array is tied to the distributed matrix <i>X</i> .
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>iwork(1)</i>	On exit, <i>iwork(1)</i> contains the minimum value of <i>liwork</i> required for optimum performance (for real flavors).
<i>rwork(1)</i>	On exit, <i>rwork(1)</i> contains the minimum value of <i>lrwork</i> required for optimum performance (for complex flavors).
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> th argument is an array and the <i>j</i> th entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>); if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?porfs

Improves the computed solution to a system of linear equations with symmetric/Hermitian positive definite distributed matrix and provides error bounds and backward error estimates for the solution.

Syntax

Fortran:

```
call psporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descraf, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)

call pdporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descraf, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, iwork, liwork, info)

call pcporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descraf, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)

call pzporfs(uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descraf, b, ib, jb, descb,
x, ix, jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)
```

C:

```
void psporfs (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT
*descraf , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *x , MKL_INT
*ix , MKL_INT *jx , MKL_INT *descx , float *ferr , float *berr , float *work , MKL_INT
*lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pdporfs (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT
*descraf , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *x , MKL_INT
*ix , MKL_INT *jx , MKL_INT *descx , double *ferr , double *berr , double *work ,
MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pcporfs (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *iaf , MKL_INT *jaf ,
MKL_INT *descraf , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *ferr , float
*berr , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT
*info );

void pzporfs (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *iaf , MKL_INT *jaf ,
MKL_INT *descraf , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *ferr , double
*berr , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT *lrwork ,
MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?porfs routine improves the computed solution to the system of linear equations

$\text{sub}(A) * \text{sub}(X) = \text{sub}(B)$,

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ is a real symmetric or complex Hermitian positive definite distributed matrix and

```
sub(B) = B(ib:ib+n-1, jb:jb+nrhs-1),
sub(X) = X(ix:ix+n-1, jx:jx+nrhs-1)
```

are right-hand side and solution submatrices, respectively. This routine also provides error bounds and backward error estimates for the solution.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(<i>A</i>) is stored. If <i>uplo</i> = 'U', sub(<i>A</i>) is upper triangular. If <i>uplo</i> = 'L', sub(<i>A</i>) is lower triangular.
<i>n</i>	(global) INTEGER. The order of the distributed matrix sub(<i>A</i>) (<i>n</i> ≥0).
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides, i.e., the number of columns of the matrices sub(<i>B</i>) and sub(<i>X</i>) (<i>nrhs</i> ≥0).
<i>a</i> , <i>af</i> , <i>b</i> , <i>x</i>	(local) REAL for psporfs DOUBLE PRECISION for pdporfs COMPLEX for pcporfs DOUBLE COMPLEX for pzporfs. Pointers into the local memory to arrays of local dimension <i>a</i> (lld_a,LOCc(ja+n-1)), <i>af</i> (lld_af,LOCc(ja+n-1)), <i>b</i> (lld_b,LOCc(jb+nrhs-1)), and <i>x</i> (lld_x,LOCc(jx+nrhs-1)), respectively. The array <i>a</i> contains the local pieces of the <i>n</i> -by- <i>n</i> symmetric/Hermitian distributed matrix sub(<i>A</i>). If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of sub(<i>A</i>) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of sub(<i>A</i>) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced. The array <i>af</i> contains the factors <i>L</i> or <i>U</i> from the Cholesky factorization sub(<i>A</i>) = <i>L</i> * <i>L</i> ^H or sub(<i>A</i>) = <i>U</i> ^H * <i>U</i> , as computed by p?potrf. On entry, the array <i>b</i> contains the local pieces of the distributed matrix of right hand sides sub(<i>B</i>). On entry, the array <i>x</i> contains the local pieces of the solution vectors sub(<i>X</i>). <i>ia</i> , <i>ja</i> (global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix sub(<i>A</i>), respectively. <i>desca</i> (global and local) INTEGER array, dimension (dlen_). The array descriptor for the distributed matrix <i>A</i> .

<i>iaf, jaf</i>	(global) INTEGER. The row and column indices in the global array <i>AF</i> indicating the first row and the first column of the submatrix sub(<i>AF</i>), respectively.
<i>descaf</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>AF</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of the submatrix sub(<i>B</i>), respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the global array <i>X</i> indicating the first row and the first column of the submatrix sub(<i>X</i>), respectively.
<i>descx</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>X</i> .
<i>work</i>	(local) REAL for psporfs DOUBLE PRECISION for pdporfs COMPLEX for pcporfs DOUBLE COMPLEX for pzporfs. The array <i>work</i> of dimension (<i>lwork</i>) is a workspace array.
<i>lwork</i>	(local) INTEGER. The dimension of the array <i>work</i> . <i>For real flavors:</i> <i>lwork</i> must be at least $lwork \geq 3 * LOCr(n + \text{mod}(ia-1, mb_a))$ <i>For complex flavors:</i> <i>lwork</i> must be at least $lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a))$
<i>iwork</i>	(local) INTEGER. Workspace array, size (<i>liwork</i>). Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The dimension of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ib-1, mb_b)).$
<i>rwork</i>	(local) REAL for pcporfs DOUBLE PRECISION for pzporfs Workspace array, size (<i>lrwork</i>). Used in complex flavors only.

lrwork (local or global) INTEGER. The dimension of the array *rwork*; used in complex flavors only. Must be at least $lrwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$.

Output Parameters

x On exit, contains the improved solution vectors.

ferr, berr REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays, dimension $LOCc(jb+nrhs-1)$ each.

The array *ferr* contains the estimated forward error bound for each solution vector of $\text{sub}(X)$.

If X_{TRUE} is the true solution corresponding to $\text{sub}(X)$, *ferr* is an estimated upper bound for the magnitude of the largest element in $(\text{sub}(X) - X_{TRUE})$ divided by the magnitude of the largest element in $\text{sub}(X)$. The estimate is as reliable as the estimate for *rcond*, and is almost always a slight overestimate of the true error.

This array is tied to the distributed matrix X .

The array *berr* contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of $\text{sub}(A)$ or $\text{sub}(B)$ that makes $\text{sub}(X)$ an exact solution). This array is tied to the distributed matrix X .

work(1) On exit, *work(1)* contains the minimum value of *lwork* required for optimum performance.

iwork(1) On exit, *iwork(1)* contains the minimum value of *liwork* required for optimum performance (for real flavors).

rwork(1) On exit, *rwork(1)* contains the minimum value of *lrwork* required for optimum performance (for complex flavors).

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*) ; if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?trrfs

Provides error bounds and backward error estimates for the solution to a system of linear equations with a distributed triangular coefficient matrix.

Syntax

Fortran:

```
call pstrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix,
jx, descx, ferr, berr, work, lwork, iwork, liwork, info)
```

```

call pdtrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix,
jx, descx, ferr, berr, work, lwork, iwork, liwork, info)

call pctrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix,
jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)

call pztrrfs(uplo, trans, diag, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, x, ix,
jx, descx, ferr, berr, work, lwork, rwork, lrwork, info)

```

C:

```

void pstrrfs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
float *ferr , float *berr , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT
*liwork , MKL_INT *info );

void pdtrrfs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , double *ferr , double *berr , double *work , MKL_INT *lwork , MKL_INT
*iwork , MKL_INT *liwork , MKL_INT *info );

void pctrrfs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , float *ferr , float *berr , MKL_Complex8 *work , MKL_INT
*lwork , float *rwork , MKL_INT *lrwork , MKL_INT *info );

void pztrrfs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , double *ferr , double *berr , MKL_Complex16 *work , MKL_INT
*lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The p?trrfs routine provides error bounds and backward error estimates for the solution to one of the systems of linear equations

```

sub(A)*sub(X) = sub(B),
sub(A)T*sub(X) = sub(B), or
sub(A)H*sub(X) = sub(B),
where sub(A) = A(ia:ia+n-1, ja:ja+n-1) is a triangular matrix,
sub(B) = B(ib:ib+n-1, jb:jb+nrhs-1), and
sub(X) = X(ix:ix+n-1, jx:jx+nrhs-1).

```

The solution matrix X must be computed by p?trtrs or some other means before entering this routine. The routine p?trrfs does not do iterative refinement because doing so cannot improve the backward error.

Input Parameters

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

If $uplo = 'U'$, $\text{sub}(A)$ is upper triangular. If $uplo = 'L'$, $\text{sub}(A)$ is lower triangular.

trans

(global) CHARACTER*1. Must be 'N' or 'T' or 'C'.

Specifies the form of the system of equations:

If $trans = 'N'$, the system has the form $\text{sub}(A) * \text{sub}(X) = \text{sub}(B)$ (No transpose);

If $trans = 'T'$, the system has the form $\text{sub}(A)^T * \text{sub}(X) = \text{sub}(B)$ (Transpose);

If $trans = 'C'$, the system has the form $\text{sub}(A)^H * \text{sub}(X) = \text{sub}(B)$ (Conjugate transpose).

diag

CHARACTER*1. Must be 'N' or 'U'.

If $diag = 'N'$, then $\text{sub}(A)$ is non-unit triangular.

If $diag = 'U'$, then $\text{sub}(A)$ is unit triangular.

n

(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).

nrhs

(global) INTEGER. The number of right-hand sides, that is, the number of columns of the matrices $\text{sub}(B)$ and $\text{sub}(X)$ ($nrhs \geq 0$).

a, b, x

(local)

REAL for pstrrfs

DOUBLE PRECISION for pdtrrfs

COMPLEX for pcrrrfs

DOUBLE COMPLEX for pztrrfs.

Pointers into the local memory to arrays of local dimension

$a(\text{lld}_a, LOCC(ja+n-1))$, $b(\text{lld}_b, LOCC(jb+nrhs-1))$, and

$x(\text{lld}_x, LOCC(jx+nrhs-1))$, respectively.

The array a contains the local pieces of the original triangular distributed matrix $\text{sub}(A)$.

If $uplo = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.

If $diag = 'U'$, the diagonal elements of $\text{sub}(A)$ are also not referenced and are assumed to be 1.

On entry, the array b contains the local pieces of the distributed matrix of right hand sides $\text{sub}(B)$.

On entry, the array x contains the local pieces of the solution vectors $\text{sub}(X)$.

ia, ja

(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of the submatrix sub(<i>B</i>), respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the global array <i>X</i> indicating the first row and the first column of the submatrix sub(<i>X</i>), respectively.
<i>descx</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>X</i> .
<i>work</i>	(local) REAL for pstrrfs DOUBLE PRECISION for pdtrrfs COMPLEX for pctrrfs DOUBLE COMPLEX for pztrrfs. The array <i>work</i> of dimension (<i>lwork</i>) is a workspace array.
<i>lwork</i>	(local) INTEGER. The dimension of the array <i>work</i> . <i>For real flavors:</i> <i>lwork</i> must be at least $lwork \geq 3 * LOCr(n + \text{mod}(ia-1, mb_a))$
	<i>For complex flavors:</i> <i>lwork</i> must be at least $lwork \geq 2 * LOCr(n + \text{mod}(ia-1, mb_a))$
<i>iwork</i>	(local) INTEGER. Workspace array, size (<i>liwork</i>). Used in real flavors only.
<i>liwork</i>	(local or global) INTEGER. The dimension of the array <i>iwork</i> ; used in real flavors only. Must be at least $liwork \geq LOCr(n + \text{mod}(ib-1, mb_b)).$
<i>rwork</i>	(local) REAL for pctrrfs DOUBLE PRECISION for pztrrfs Workspace array, size (<i>lrwork</i>). Used in complex flavors only.
<i>lrwork</i>	(local or global) INTEGER. The dimension of the array <i>rwork</i> ; used in complex flavors only. Must be at least $lrwork \geq LOCr(n + \text{mod}(ib-1, mb_b))$.

Output Parameters

<i>ferr, berr</i>	REAL for single precision flavors.
-------------------	------------------------------------

DOUBLE PRECISION for double precision flavors.

Arrays, dimension $\text{LOCc}(jb+nrhs-1)$ each.

The array $ferr$ contains the estimated forward error bound for each solution vector of $\text{sub}(X)$.

If $XTRUE$ is the true solution corresponding to $\text{sub}(X)$, $ferr$ is an estimated upper bound for the magnitude of the largest element in $(\text{sub}(X) - XTRUE)$ divided by the magnitude of the largest element in $\text{sub}(X)$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.

This array is tied to the distributed matrix X .

The array $berr$ contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of $\text{sub}(A)$ or $\text{sub}(B)$ that makes $\text{sub}(X)$ an exact solution). This array is tied to the distributed matrix X .

$work(1)$

On exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance.

$iwork(1)$

On exit, $iwork(1)$ contains the minimum value of $liwork$ required for optimum performance (for real flavors).

$rwork(1)$

On exit, $rwork(1)$ contains the minimum value of $lrwork$ required for optimum performance (for complex flavors).

$info$

(global) INTEGER. If $info=0$, the execution is successful.

$info < 0$:

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Routines for Matrix Inversion

This section describes ScaLAPACK routines that compute the inverse of a matrix based on the previously obtained factorization. Note that it is not recommended to solve a system of equations $Ax = b$ by first computing A^{-1} and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see [Solving Systems of Linear Equations](#)); this is more efficient and more accurate.

p?getri

Computes the inverse of a LU-factored distributed matrix.

Syntax

Fortran:

```
call psgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pdgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pcgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
call pzgetri(n, a, ia, ja, desca, ipiv, work, lwork, iwork, liwork, info)
```

C:

```

void psgetri (MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_INT *ipiv , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *info );

void pdgetri (MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_INT *ipiv , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *info );

void pcgetri (MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *ipiv , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *iwork ,
MKL_INT *liwork , MKL_INT *info );

void pzgetri (MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *ipiv , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *iwork ,
MKL_INT *liwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The p?getri routine computes the inverse of a general distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the LU factorization computed by p?getrf. This method inverts U and then computes the inverse of $\text{sub}(A)$ by solving the system

```
inv(sub(A)) * L = inv(U)
```

```
for inv(sub(A)).
```

Input Parameters

<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) REAL for psgetri DOUBLE PRECISION for pdgetri COMPLEX for pcgetri DOUBLE COMPLEX for pzgetri. Pointer into the local memory to an array of local dimension $a(lld_a, LOCo(ja+n-1))$. On entry, the array <i>a</i> contains the local pieces of the L and U obtained by the factorization $\text{sub}(A) = P^* L^* U$ computed by p?getrf.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the global array <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for psgetri DOUBLE PRECISION for pdgetri

COMPLEX for pcgetri

DOUBLE COMPLEX for pzgetri.

The array *work* of dimension (*lwork*) is a workspace array.

lwork

(local) INTEGER. The dimension of the array *work*. *lwork* must be at least

$lwork \geq LOCr(n + \text{mod}(ia-1, mb_a)) * nb_a$.

The array *work* is used to keep at most an entire column block of sub(*A*).

iwork

(local) INTEGER. Workspace array used for physically transposing the pivots, size (*liwork*).

liwork

(local or global) INTEGER. The dimension of the array *iwork*.

The minimal value *liwork* of is determined by the following code:

```
if NPROW == NPCOL then
```

```
    liwork = LOCc(n_a + mod(ja-1, nb_a)) + nb_a
```

```
else
```

```
    liwork = LOCc(n_a + mod(ja-1, nb_a)) +
```

```
    max(ceil(ceil(LOCr(m_a)/mb_a)/(lcm/NPROW)), nb_a)
```

```
end if
```

where *lcm* is the least common multiple of process rows and columns (NPROW and NPCOL).

Output Parameters

ipiv

(local) INTEGER.

Array, dimension ($LOCr(m_a) + mb_a$).

This array contains the pivoting information.

If $ipiv(i)=j$, then the local row *i* was swapped with the global row *j*.

This array is tied to the distributed matrix *A*.

work(1)

On exit, *work*(1) contains the minimum value of *lwork* required for optimum performance.

iwork(1)

On exit, *iwork*(1) contains the minimum value of *liwork* required for optimum performance.

info

(global) INTEGER. If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then $info = -(i*100+j)$; if the *i*-th argument is a scalar and had an illegal value, then $info = -i$.

info > 0:

If *info* = *i*, $U(i,i)$ is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

p?potri

Computes the inverse of a symmetric/Hermitian positive definite distributed matrix.

Syntax**Fortran:**

```
call pspotri(uplo, n, a, ia, ja, desca, info)
call pdpotri(uplo, n, a, ia, ja, desca, info)
call pcpotri(uplo, n, a, ia, ja, desca, info)
call pzpotri(uplo, n, a, ia, ja, desca, info)
```

C:

```
void pspotri (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *info );
void pdpotri (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
void pcpotri (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
void pzpotri (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?potri` routine computes the inverse of a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the Cholesky factorization $\text{sub}(A) = U^H \cdot U$ or $\text{sub}(A) = L \cdot L^H$ computed by `p?potrf`.

Input Parameters

<code>uplo</code>	(global) CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '.
	Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored.
	If <code>uplo</code> = ' <code>U</code> ', upper triangle of $\text{sub}(A)$ is stored. If <code>uplo</code> = ' <code>L</code> ', lower triangle of $\text{sub}(A)$ is stored.
<code>n</code>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<code>a</code>	(local) REAL for <code>pspotri</code> DOUBLE PRECISION for <code>pdpotri</code> COMPLEX for <code>pcpotri</code> DOUBLE COMPLEX for <code>pzpotri</code> .

Pointer into the local memory to an array of local dimension $a(1\text{ld_}a, LOCc(ja+n-1))$.

On entry, the array a contains the local pieces of the triangular factor U or L from the Cholesky factorization $\text{sub}(A) = U^H * U$, or $\text{sub}(A) = L * L^H$, as computed by p?potrf.

ia, ja

(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

$desca$

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .

Output Parameters

a

On exit, overwritten by the local pieces of the upper or lower triangle of the (symmetric/Hermitian) inverse of $\text{sub}(A)$.

$info$

(global) INTEGER. If $info=0$, the execution is successful.

$info < 0$:

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

$info > 0$:

If $info = i$, the (i, i) element of the factor U or L is zero, and the inverse could not be computed.

p?trtri

Computes the inverse of a triangular distributed matrix.

Syntax

Fortran:

```
call pstrtri(uplo, diag, n, a, ia, ja, desca, info)
call pdtrtri(uplo, diag, n, a, ia, ja, desca, info)
call pctrtri(uplo, diag, n, a, ia, ja, desca, info)
call pztrtri(uplo, diag, n, a, ia, ja, desca, info)
```

C:

```
void pstrtri (char *uplo , char *diag , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , MKL_INT *info );
void pdtrtri (char *uplo , char *diag , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , MKL_INT *info );
void pctrtri (char *uplo , char *diag , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
void pztrtri (char *uplo , char *diag , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?trtri` routine computes the inverse of a real or complex upper or lower triangular distributed matrix
 $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

<code>uplo</code>	(global) CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '. Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular. If <code>uplo</code> = ' <code>U</code> ', $\text{sub}(A)$ is upper triangular. If <code>uplo</code> = ' <code>L</code> ', $\text{sub}(A)$ is lower triangular.
<code>diag</code>	CHARACTER*1. Must be ' <code>N</code> ' or ' <code>U</code> '. Specifies whether or not the distributed matrix $\text{sub}(A)$ is unit triangular. If <code>diag</code> = ' <code>N</code> ', then $\text{sub}(A)$ is non-unit triangular. If <code>diag</code> = ' <code>U</code> ', then $\text{sub}(A)$ is unit triangular.
<code>n</code>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<code>a</code>	(local) REAL for <code>pstrtri</code> DOUBLE PRECISION for <code>pdtrtri</code> COMPLEX for <code>pctrtri</code> DOUBLE COMPLEX for <code>pztrtri</code> . Pointer into the local memory to an array of local dimension $a(1\text{ld}_a, LOCc(ja+n-1))$. The array <code>a</code> contains the local pieces of the triangular distributed matrix $\text{sub}(A)$. If <code>uplo</code> = ' <code>U</code> ', the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular matrix to be inverted, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced. If <code>uplo</code> = ' <code>L</code> ', the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<code>desca</code>	(global and local) INTEGER array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix A .

Output Parameters

a On exit, overwritten by the (triangular) inverse of the original matrix.

info (global) INTEGER. If *info*=0, the execution is successful.

info < 0:
If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:
If *info* = *k*, *A*(*ia*+*k*-1, *ja*+*k*-1) is exactly zero. The triangular matrix sub(*A*) is singular and its inverse can not be computed.

Routines for Matrix Equilibration

ScalAPACK routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

p?geequ

Computes row and column scaling factors intended to equilibrate a general rectangular distributed matrix and reduce its condition number.

Syntax

Fortran:

```
call psgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pdgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pcgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
call pzgeequ(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, info)
```

C:

```
void psgeequ (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *r , float *c , float *rowcnd , float *colcnd , float *amax , MKL_INT
*info );
void pdgeequ (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *r , double *c , double *rowcnd , double *colcnd , double
*amax , MKL_INT *info );
void pcgeequ (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *r , float *c , float *rowcnd , float *colcnd , float *amax ,
MKL_INT *info );
void pzgeequ (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *r , double *c , double *rowcnd , double *colcnd , double
*amax , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?geequ` routine computes row and column scalings intended to equilibrate an m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ and reduce its condition number. The output array r returns the row scale factors and the array c the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements $b_{ij}=r(i)*a_{ij}*c(j)$ have absolute value 1.

$r(i)$ and $c(j)$ are restricted to be between $SMLNUM$ = smallest safe number and $BIGNUM$ = largest safe number. Use of these scaling factors is not guaranteed to reduce the condition number of $\text{sub}(A)$ but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the `?lamch` routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

The auxiliary function `p?laqge` uses scaling factors computed by `p?geequ` to scale a general rectangular matrix.

Input Parameters

m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$ ($m \geq 0$).
n	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) REAL for <code>psgeequ</code> DOUBLE PRECISION for <code>pdgeequ</code> COMPLEX for <code>pcgeequ</code> DOUBLE COMPLEX for <code>pzgeequ</code> . Pointer into the local memory to an array of local dimension $a(1\text{ld}_a, LOCc(ja+n-1))$. The array a contains the local pieces of the m -by- n distributed matrix whose equilibration factors are to be computed.
ia, ja	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
$desca$	(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .

Output Parameters

r, c	(local) REAL for single precision flavors; DOUBLE PRECISION for double precision flavors. Arrays, dimension $LOCr(m_a)$ and $LOCr(n_a)$, respectively. If $info = 0$, or $info > ia+m-1$, the array r ($ia:ia+m-1$) contains the row scale factors for $\text{sub}(A)$. r is aligned with the distributed matrix A , and replicated across every process column. r is tied to the distributed matrix A .
--------	---

If $info = 0$, the array $c(ja:ja+n-1)$ contains the column scale factors for $\text{sub}(A)$. c is aligned with the distributed matrix A , and replicated down every process row. c is tied to the distributed matrix A .

$\text{rowcnd}, \text{colcnd}$

(global) REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

If $info = 0$ or $info > ia+m-1$, rowcnd contains the ratio of the smallest $r(i)$ to the largest $r(i)$ ($ia \leq i \leq ia+m-1$). If $\text{rowcnd} \geq 0.1$ and amax is neither too large nor too small, it is not worth scaling by $r(ia:ia+m-1)$.

If $info = 0$, colcnd contains the ratio of the smallest $c(j)$ to the largest $c(j)$ ($ja \leq j \leq ja+n-1$).

If $\text{colcnd} \geq 0.1$, it is not worth scaling by $c(ja:ja+n-1)$.

amax

(global) REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest matrix element. If amax is very close to overflow or very close to underflow, the matrix should be scaled.

info

(global) INTEGER. If $\text{info}=0$, the execution is successful.

$\text{info} < 0$:

If the i th argument is an array and the j th entry had an illegal value, then $\text{info} = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.

$\text{info} > 0$:

If $\text{info} = i$ and

$i \leq m$, the i th row of the distributed matrix

$\text{sub}(A)$ is exactly zero;

$i > m$, the $(i-m)$ th column of the distributed

matrix $\text{sub}(A)$ is exactly zero.

p?poequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite distributed matrix and reduce its condition number.

Syntax

Fortran:

```
call pspoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pdpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pcpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
call pzpoequ(n, a, ia, ja, desca, sr, sc, scond, amax, info)
```

C:

```

void pspoequ (MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
float *sr , float *sc , float *scond , float *amax , MKL_INT *info );
void pdpoequ (MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
double *sr , double *sc , double *scond , double *amax , MKL_INT *info );
void pcpoequ (MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *sr , float *sc , float *scond , float *amax , MKL_INT *info );
void pzpoequ (MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , double *sr , double *sc , double *scond , double *amax , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The `p?poequ` routine computes row and column scalings intended to equilibrate a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ and reduce its condition number (with respect to the two-norm). The output arrays `sr` and `sc` return the row and column scale factors

.....

These factors are chosen so that the scaled distributed matrix B with elements $b_{ij}=s(i)*a_{ij}*s(j)$ has ones on the diagonal.

This choice of `sr` and `sc` puts the condition number of B within a factor n of the smallest possible condition number over all possible diagonal scalings.

The auxiliary function `p?laqsy` uses scaling factors computed by `p?geequ` to scale a general rectangular matrix.

Input Parameters

<code>n</code>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<code>a</code>	(local) REAL for <code>pspoequ</code> DOUBLE PRECISION for <code>pdpoequ</code> COMPLEX for <code>pcpoequ</code> DOUBLE COMPLEX for <code>pzpoequ</code> . Pointer into the local memory to an array of local dimension $a(1:l_d_a, LOCc(ja+n-1))$.
	The array <code>a</code> contains the n -by- n symmetric/Hermitian positive definite distributed matrix $\text{sub}(A)$ whose scaling factors are to be computed. Only the diagonal elements of $\text{sub}(A)$ are referenced.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

sr, sc

(local)

REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

Arrays, dimension *LOCr(m_a)* and *LOCc(n_a)*, respectively.

If *info* = 0, the array *sr(ia:ia+n-1)* contains the row scale factors for sub(*A*). *sr* is aligned with the distributed matrix *A*, and replicated across every process column. *sr* is tied to the distributed matrix *A*.

If *info* = 0, the array *sc(ja:ja+n-1)* contains the column scale factors for sub(*A*). *sc* is aligned with the distributed matrix *A*, and replicated down every process row. *sc* is tied to the distributed matrix *A*.

scond

(global)

REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

If *info* = 0, *scond* contains the ratio of the smallest *sr(i)* (or *sc(j)*) to the largest *sr(i)* (or *sc(j)*), with

ia≤*i*≤*ia+n-1* and *ja*≤*j*≤*ja+n-1*.

If *scond* ≥ 0.1 and *amax* is neither too large nor too small, it is not worth scaling by *sr* (or *sc*).

amax

(global)

REAL for single precision flavors;

DOUBLE PRECISION for double precision flavors.

Absolute value of the largest matrix element. If *amax* is very close to overflow or very close to underflow, the matrix should be scaled.

info

(global) INTEGER.

If *info*=0, the execution is successful.

info < 0:

If the *i*-th argument is an array and the *j*-th entry had an illegal value, then *info* = -(*i**100+*j*) ; if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

info > 0:

If *info* = *k*, the *k*-th diagonal entry of sub(*A*) is nonpositive.

Orthogonal Factorizations

This section describes the ScalAPACK routines for the *QR* (*RQ*) and *LQ* (*QL*) factorization of matrices. Routines for the *RZ* factorization as well as for generalized *QR* and *RQ* factorizations are also included. For the mathematical definition of the factorizations, see the respective LAPACK sections or refer to [SLUG].

Table "Computational Routines for Orthogonal Factorizations" lists ScalAPACK routines that perform orthogonal factorization of matrices.

Computational Routines for Orthogonal Factorizations

Matrix type, factorization	Factorize without pivoting	Factorize with pivoting	Generate matrix Q	Apply matrix Q
general matrices, QR factorization	p?geqrf	p?geqpf	p?orgqr p?ungqr	p?ormqr p?unmqr
general matrices, RQ factorization	p?gerqf		p?orgrq p?ungrq	p?ormrq p?unmrq
general matrices, LQ factorization	p?gelqf		p?orglq p?unglq	p?ormlq p?unmlq
general matrices, QL factorization	p?geqlf		p?orgql p?ungql	p?ormql p?unmql
trapezoidal matrices, RZ factorization	p?tzrzf			p?ormrz p?unmrz
pair of matrices, generalized QR factorization	p?ggqrf			
pair of matrices, generalized RQ factorization	p?ggrqf			

p?geqrf

Computes the QR factorization of a general *m*-by-*n* matrix.

Syntax

Fortran:

```
call psgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqrf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psgeqrf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdgeqrf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
```

```
void pcgeqrf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

void pzgeqrf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?geqrf routine forms the QR factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$$A = Q \cdot R$$

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the distributed submatrix $\text{sub}(A)$; ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed submatrix $\text{sub}(A)$; ($n \geq 0$).
<i>a</i>	(local) REAL for psgeqrf DOUBLE PRECISION for pdgeqrf COMPLEX for pcgeqrf DOUBLE COMPLEX for pzgeqrf. Pointer into the local memory to an array of local dimension (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>
<i>work</i>	(local). REAL for psgeqrf DOUBLE PRECISION for pdgeqrf. COMPLEX for pcgeqrf. DOUBLE COMPLEX for pzgeqrf Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq nb_a * (mp0 + nq0 + nb_a)$, where

```

iroff = mod(ia-1, mb_a), icooff = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mp0 = numroc(m+iroff, mb_a, MYROW, iarow, NPROW),
nq0 = numroc(n+icooff, nb_a, MYCOL, iacol, NPCOL), and numroc,
indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL
can be determined by calling the subroutine blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>a</i>	The elements on and above the diagonal of sub(<i>A</i>) contain the min(<i>m,n</i>)-by- <i>n</i> upper trapezoidal matrix <i>R</i> (<i>R</i> is upper triangular if <i>m</i> ≥ <i>n</i>); the elements below the diagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>tau</i>	(local) REAL for psgeqrf DOUBLE PRECISION for pdgeqrf COMPLEX for pcgeqrf DOUBLE COMPLEX for pzgeqrf. Array, size <i>LOCc</i> (ja+min(<i>m,n</i>)-1). Contains the scalar factor <i>tau</i> of elementary reflectors. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0, the execution is successful. < 0, if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (i* 100+j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

$$Q = H(ja) * H(ja+1) * \dots * H(ja+k-1),$$

where *k* = min(*m,n*).

Each *H(i)* has the form

$$H(i) = I - tau * v * v'$$

where τ_{ii} is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$, and τ_{ii} in $\tau_{ii}(ja+i-1)$.

p?geqpf

Computes the QR factorization of a general m-by-n matrix with pivoting.

Syntax

Fortran:

```
call psgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
call pdgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
call pcgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
call pzgeqpf(m, n, a, ia, ja, desca, ipiv, tau, work, lwork, info)
```

C:

```
void psgeqpf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja ,
              MKL_INT *desca , MKL_INT *ipiv , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdgeqpf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
              MKL_INT *desca , MKL_INT *ipiv , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcgeqpf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
              MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
              *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *info );
void pzgeqpf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
              MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
              *lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?geqpf routine forms the QR factorization with column pivoting of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$\text{sub}(A) * P = Q * R$

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the submatrix $\text{sub}(A)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) REAL for psgeqpf DOUBLE PRECISION for pdgeqpf COMPLEX for pcgeqpf DOUBLE COMPLEX for pzgeqpf.

Pointer into the local memory to an array of local dimension ($l1d_a$, $LOCc(ja+n-1)$).

Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.

ia, ja
(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.

desca
(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix *A*.

work
(local).

REAL for *psgeqpf*
DOUBLE PRECISION for *pdgeqpf*.
COMPLEX for *pcgeqpf*.
DOUBLE COMPLEX for *pzgeqpf*

Workspace array of dimension *lwork*.

lwork
(local or global) INTEGER, dimension of *work*, must be at least

For real flavors:

$$lwork \geq \max(3, mp0 + nq0) + LOCc(ja+n-1) + nq0.$$

For complex flavors:

$$lwork \geq \max(3, mp0 + nq0) .$$

Here

```
iroff = mod(ia-1, mb_a), icoff = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mp0 = numroc(m+iroff, mb_a, MYROW, iarow, NPROW),
nq0 = numroc(n+icoff, nb_a, MYCOL, iacol, NPCOL),
LOCc(ja+n-1) = numroc(ja+n-1, nb_a, MYCOL, csrc_a, NPCOL),
and numroc, indxg2p are ScaLAPACK tool functions.
```

You can determine MYROW, MYCOL, NPROW and NPCOL by calling the blacs_gridinfo subroutine.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a
The elements on and above the diagonal of $\text{sub}(A)$ contain the $\min(m, n)$ -by- n upper trapezoidal matrix *R* (*R* is upper triangular if $m \geq n$); the elements below the diagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

<i>ipiv</i>	(local) INTEGER. Array, size $LOCc(ja+n-1)$. $ipiv(i) = k$, the local i -th column of $\text{sub}(A)*P$ was the global k -th column of $\text{sub}(A)$. <i>ipiv</i> is tied to the distributed matrix A .
<i>tau</i>	(local) REAL for psgeqpf DOUBLE PRECISION for pdgeqpf COMPLEX for pcgeqpf DOUBLE COMPLEX for pzgeqpf. Array, size $LOCc(ja+\min(m, n)-1)$. Contains the scalar factor <i>tau</i> of elementary reflectors. <i>tau</i> is tied to the distributed matrix A .
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0, the execution is successful. < 0, if the i -th argument is an array and the j -entry had an illegal value, then <i>info</i> = $-(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(1)*H(2)*\dots*H(k)$$

where $k = \min(m, n)$.

Each $H(i)$ has the form

$$H = I - tau*v*v'$$

where *tau* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$.

The matrix P is represented in *ipiv* as follows: if $ipiv(j)=i$ then the j -th column of P is the i -th canonical unit vector.

p?orgqr

Generates the orthogonal matrix Q of the QR factorization formed by p?geqrf.

Syntax

Fortran:

```
call psorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psorgqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
```

```
void pdorgqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?orgqr` routine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by `p?geqrf`.

Input Parameters

<code>m</code>	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$ ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ ($m \geq n \geq 0$).
<code>k</code>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<code>a</code>	(local) REAL for <code>psorgqr</code> DOUBLE PRECISION for <code>pdorgqr</code> Pointer into the local memory to an array of local dimension $(\text{lld}_a, \text{LOCc}(ja+n-1))$. The j -th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja + k - 1$, as returned by <code>p?geqrf</code> in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<code>desca</code>	(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .
<code>tau</code>	(local) REAL for <code>psorgqr</code> DOUBLE PRECISION for <code>pdorgqr</code> Array, size $\text{LOCc}(ja+k-1)$. Contains the scalar factor τ_j of elementary reflectors $H(j)$ as returned by <code>p?geqrf</code> . τ_j is tied to the distributed matrix A .
<code>work</code>	(local) REAL for <code>psorgqr</code> DOUBLE PRECISION for <code>pdorgqr</code>

Workspace array of dimension of *lwork*.

lwork

(local or global) INTEGER, dimension of *work*.

Must be at least $lwork \geq nb_a * (nqa0 + mpa0 + nb_a)$, where

$iroffa = \text{mod}(ia-1, mb_a)$, $icoffa = \text{mod}(ja-1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW})$,

$iacol = \text{indxg2p}(ja, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL})$,

$mpa0 = \text{numroc}(m+iroffa, mb_a, \text{MYROW}, iarow, \text{NPROW})$,

$nqa0 = \text{numroc}(n+icoffa, nb_a, \text{MYCOL}, iacol, \text{NPCOL})$;

indxg2p and *numroc* are ScalAPACK tool functions; *MYROW*, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a

Contains the local pieces of the *m*-by-*n* distributed matrix Q.

work(1)

On exit, *work(1)* contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+j), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ungqr

Generates the complex unitary matrix Q of the QR factorization formed by p?geqrf.

Syntax

Fortran:

```
call pcungqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pzungqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void pcungqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
```

```
void pzungqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This routine generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by [p?geqrf](#).

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$; ($m \geq 0$).
n	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ ($m \geq n \geq 0$).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
a	<p>(local)</p> <p>COMPLEX for pcungqr</p> <p>DOUBLE COMPLEX for pzungqr</p> <p>Pointer into the local memory to an array of dimension ($lld_a, LOCc(ja+n-1)$). The j-th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqrf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$.</p>
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .
tau	<p>(local)</p> <p>COMPLEX for pcungqr</p> <p>DOUBLE COMPLEX for pzungqr</p> <p>Array, size $LOCc(ja+k-1)$.</p> <p>Contains the scalar factor τ_j of elementary reflectors $H(j)$ as returned by p?geqrf. τ_j is tied to the distributed matrix A.</p>
$work$	<p>(local)</p> <p>COMPLEX for pcungqr</p> <p>DOUBLE COMPLEX for pzungqr</p> <p>Workspace array of dimension of $lwork$.</p>
$lwork$	(local or global) INTEGER, dimension of $work$, must be at least $lwork \geq nb_a * (nqa0 + mpa0 + nb_a)$, where

```

iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL)

```

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork` = -1, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pixerbla`.

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q .
<code>work(1)</code>	On exit <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then <code>info</code> = $-(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then <code>info</code> = $-i$.

p?ormqr

Multiples a general matrix by the orthogonal matrix Q of the QR factorization formed by p?geqrf.

Syntax

Fortran:

```

call psormqr(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pdormqr(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

```

C:

```

void psormqr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );

void pdormqr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );

```

Include Files

- C: `mkl_scalapack.h`

Description

The `p?ormqr` routine overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	<code>side = 'L'</code>	<code>side = 'R'</code>
<code>trans = 'N':</code>	$Q * \text{sub}(C)$	$\text{sub}(C) * Q$
<code>trans = 'T':</code>	$Q^T * \text{sub}(C)$	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1) H(2) \dots H(k)$$

as returned by `p?geqrf`. Q is of order m if `side = 'L'` and of order n if `side = 'R'`.

Input Parameters

<code>side</code>	(global) CHARACTER
	<code>= 'L':</code> Q or Q^T is applied from the left.
	<code>= 'R':</code> Q or Q^T is applied from the right.
<code>trans</code>	(global) CHARACTER
	<code>= 'N'</code> , no transpose, Q is applied.
	<code>= 'T'</code> , transpose, Q^T is applied.
<code>m</code>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<code>k</code>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <code>side = 'L'</code> , $m \geq k \geq 0$ If <code>side = 'R'</code> , $n \geq k \geq 0$.
<code>a</code>	(local) REAL for <code>psormqr</code> DOUBLE PRECISION for <code>pdormqr</code> . Pointer into the local memory to an array of dimension (<code>lld_a</code> , <code>LOCc(ja+k-1)</code>). The j -th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by <code>p?geqrf</code> in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If <code>side = 'L'</code> , $lld_a \geq \max(1, LOCr(ia+m-1))$ If <code>side = 'R'</code> , $lld_a \geq \max(1, LOCr(ia+n-1))$
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr Array, size $LOC_C(ja+k-1)$. Contains the scalar factor <i>tau</i> (<i>j</i>) of elementary reflectors $H(j)$ as returned by p?geqrf. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr Pointer into the local memory to an array of local dimension (<i>lld_c</i> , $LOC_C(jc+n-1)$). Contains the local pieces of the distributed matrix <i>sub(C)</i> to be factored.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descC</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr. Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: <i>if side = 'L',</i> <i>lwork ≥ max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a</i> <i>else if side = 'R',</i> <i>lwork ≥ max((nb_a * (nb_a - 1)) / 2, (nqc0 + max(npa0 + numroc(numroc(n + ioffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0)) * nb_a) + nb_a * nb_a</i> <i>end if</i> <i>where</i> <i>lcmq = lcm/NPCOL with lcm = ilcm(NPROW, NPCOL),</i> <i>iroffa = mod(ia-1, mb_a),</i> <i>icoffa = mod(ja-1, nb_a),</i> <i>iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),</i> <i>npoa0 = numroc(n + iroffa, mb_a, MYROW, iarow, NPROW),</i> <i>iroffc = mod(ic-1, mb_c),</i>

```

    icooffc = mod(jc-1, nb_c),
    icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
    iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
    mpc0= numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
    nqc0= numroc(n+icooffc, nb_c, MYCOL, iccol, NPCOL),
    ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL,
    NPROW and NPCOL can be determined by calling the subroutine
    blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pixerbla*.

Output Parameters

<i>c</i>	Overwritten by the product $Q^* \text{sub}(C)$, or $Q^T * \text{sub}(C)$, or $\text{sub}(C) * Q^T$, or $\text{sub}(C) * Q$.
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (i* 100+j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?unmqr

Multiples a complex matrix by the unitary matrix Q of the QR factorization formed by p?geqrf.

Syntax

Fortran:

```

call pcunmqr(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pzunmqr(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

```

C:

```

void pcunmqr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunmqr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

This routine overwrites the general complex m -by- n distributed matrix sub $(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q^* \text{sub}(C)$
<i>trans</i> = 'T':	$Q^H \text{sub}(C)$
	$\text{sub}(C)^* Q$
	$\text{sub}(C)^* Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(1) H(2) \dots H(k)$ as returned by [p?geqrf](#). Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) CHARACTER
= 'L':	Q or Q^H is applied from the left.
= 'R':	Q or Q^H is applied from the right.
<i>trans</i>	(global) CHARACTER
= 'N',	no transpose, Q is applied.
= 'C',	conjugate transpose, Q^H is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) COMPLEX for pcunmqr DOUBLE COMPLEX for pzunmqr . Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+k-1)</i>). The j -th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja + k - 1$, as returned by p?geqrf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If <i>side</i> = 'L', <i>lld_a</i> $\geq \max(1, LOCr(ia+m-1))$ If <i>side</i> = 'R', <i>lld_a</i> $\geq \max(1, LOCr(ia+n-1))$

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for <code>pcunmqr</code> DOUBLE COMPLEX for <code>pzunmqr</code> Array, size $LOC_C(ja+k-1)$. Contains the scalar factor <i>tau</i> (<i>j</i>) of elementary reflectors $H(j)$ as returned by <code>p?geqrf</code> . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) COMPLEX for <code>pcunmqr</code> DOUBLE COMPLEX for <code>pzunmqr</code> . Pointer into the local memory to an array of local dimension (<i>lld_c</i> , $LOC_C(jc+n-1)$). Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descC</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) COMPLEX for <code>pcunmqr</code> DOUBLE COMPLEX for <code>pzunmqr</code> . Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: If <i>side</i> = 'L', $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a$ else if <i>side</i> = 'R', $lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max(npa0 + numroc(numroc(n+icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0)) * nb_a) + nb_a * nb_a$ end if where $lcmq = lcm/NPCOL \text{ with } lcm = \text{ilcm}(NPROW, NPCOL),$ $iroffa = \text{mod}(ia-1, mb_a),$

```

        icooffa = mod(ja-1, nb_a),
        iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
        npa0 = numroc(n+iroffa, mb_a, MYROW, iarow, NPROW),
        iroffc = mod(ic-1, mb_c),
        icooffc = mod(jc-1, nb_c),
        icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
        iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
        mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
        nqc0 = numroc(n+icooffc, nb_c, MYCOL, iccol, NPCOL),
        ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL,
        NPROW and NPCOL can be determined by calling the subroutine
        blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

<i>c</i>	Overwritten by the product $Q^* \text{sub}(C)$, or $Q^H * \text{sub}(C)$, or $\text{sub}(C) * Q^H$, or $\text{sub}(C) * Q$.
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (i * 100+j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = -i.

p?gelqf

Computes the LQ factorization of a general rectangular matrix.

Syntax

Fortran:

```

call psgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgelqf(m, n, a, ia, ja, desca, tau, work, lwork, info)

```

C:

```

void psgelqf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );

```

```

void pdgelqf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcgelqf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );
void pzgelqf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );

```

Include Files

- C: mkl_scalapack.h

Description

The p?gelqf routine computes the LQ factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ia:ia+n-1) = L*Q$.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<i>a</i>	(local) REAL for psgelqf DOUBLE PRECISION for pdgelqf COMPLEX for pcgelqf DOUBLE COMPLEX for pzgelqf Pointer into the local memory to an array of local dimension (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ia:ia+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for psgelqf DOUBLE PRECISION for pdgelqf COMPLEX for pcgelqf DOUBLE COMPLEX for pzgelqf Workspace array of dimension <i>lwork</i> .

lwork (local or global) INTEGER, dimension of *work*, must be at least $lwork \geq mb_a^* (mp0 + nq0 + mb_a)$, where

$$\begin{aligned} iroff &= \text{mod}(ia-1, mb_a), \\ icoff &= \text{mod}(ja-1, nb_a), \\ iarow &= \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}), \\ iacol &= \text{indxg2p}(ja, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL}), \\ mp0 &= \text{numroc}(m+iroff, mb_a, \text{MYROW}, iarow, \text{NPROW}), \\ nq0 &= \text{numroc}(n+icooff, nb_a, \text{MYCOL}, iacol, \text{NPCOL}) \end{aligned}$$

`indxg2p` and `numroc` are ScalAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a The elements on and below the diagonal of $\text{sub}(A)$ contain the m by $\min(m, n)$ lower trapezoidal matrix L (L is lower trapezoidal if $m \leq n$); the elements above the diagonal, with the array *tau*, represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

tau (local)
 REAL for `psgelqf`
 DOUBLE PRECISION for `pdgelqf`
 COMPLEX for `pcgelqf`
 DOUBLE COMPLEX for `pzgelqf`
 Array, size $LOCr(ia+\min(m, n)-1)$.
 Contains the scalar factors of elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1) On exit, *work(1)* contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
 = 0: the execution is successful.
 < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = $-(i * 100 + j)$, if the *i*-th argument is a scalar and had an illegal value, then *info* = $-i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ia+k-1) * H(ia+k-2) * \dots * H(ia),$$

where $k = \min(m, n)$

Each $H(i)$ has the form

$$H(i) = I - \tau v v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$, and τ in $\tau(ia+i-1)$.

p?orglq

Generates the real orthogonal matrix Q of the LQ factorization formed by p?gelqf.

Syntax

Fortran:

```
call psorglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psorglq (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdorglq (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?orglq routine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = H(k) * \dots * H(2) * H(1)$$

as returned by p?gelqf.

Input Parameters

m (global) INTEGER. The number of rows in the submatrix sub(Q); ($m \geq 0$).

n (global) INTEGER. The number of columns in the submatrix sub(Q) ($n \geq m \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).

a (local)

REAL for psorglq

DOUBLE PRECISION for pdorglq

Pointer into the local memory to an array of local dimension (lld_a , $LOCc(ja+n-1)$). On entry, the i -th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local) REAL for psorglq DOUBLE PRECISION for pdorglq Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq mb_a^* (mpa0 + nqa0 + mb_a)$, where $iroffa = \text{mod}(ia-1, mb_a),$ $icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}),$ $iacol = \text{indxg2p}(ja, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL}),$ $mpa0 = \text{numroc}(m+iroffa, mb_a, \text{MYROW}, iarow, \text{NPROW}),$ $nqa0 = \text{numroc}(n+icoffa, nb_a, \text{MYCOL}, iacol, \text{NPCOL})$ <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla .

Output Parameters

<i>a</i>	Contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix <i>Q</i> to be factored.
<i>tau</i>	(local) REAL for psorglq DOUBLE PRECISION for pdorglq Array, size <i>LOCr</i> (<i>ia+k</i> - 1). Contains the scalar factors <i>tau</i> of elementary reflectors <i>H(i)</i> . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> * 100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?unqlq

Generates the unitary matrix Q of the LQ factorization formed by [p?gelqf](#).

Syntax**Fortran:**

```
call pcunglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzunglq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void pcunglq (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
void pzunglq (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This routine generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = (H(k))^H \dots * (H(2))^H * (H(1))^H \text{ as returned by } \text{p?gelqf}.$$

Input Parameters

m (global) INTEGER. The number of rows in the submatrix sub(Q) ($m \geq 0$).

n (global) INTEGER. The number of columns in the submatrix sub(Q) ($n \geq m \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).

a (local)

COMPLEX for pcunglq
DOUBLE COMPLEX for pzunglq

Pointer into the local memory to an array of local dimension (*lld_a*, *LOCc(ja+n-1)*). On entry, the i -th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by [p?gelqf](#) in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.

<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for <code>pcunglq</code> DOUBLE COMPLEX for <code>pzunglq</code> Array, size $LOCr(ia+k-1)$. Contains the scalar factors <i>tau</i> of elementary reflectors $H(i)$. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) COMPLEX for <code>pcunglq</code> DOUBLE COMPLEX for <code>pzunglq</code> Workspace array of dimension of <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq mb_a * (mpa0 + nqa0 + mb_a)$, where $iroffa = \text{mod}(ia-1, mb_a),$ $icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}),$ $iacol = \text{indxg2p}(ja, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL}),$ $mpa0 = \text{numroc}(m+iroffa, mb_a, \text{MYROW}, iarow, \text{NPROW}),$ $nqa0 = \text{numroc}(n+icoffa, nb_a, \text{MYCOL}, iacol, \text{NPCOL})$ <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; <i>MYROW</i> , <i>MYCOL</i> , <i>NPROW</i> and <i>NPCOL</i> can be determined by calling the subroutine <code>blacs_gridinfo</code> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <code>pxerbla</code> .

Output Parameters

<i>a</i>	Contains the local pieces of the m -by- n distributed matrix <i>Q</i> to be factored.
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?ormlq

Multiples a general matrix by the orthogonal matrix Q of the LQ factorization formed by p?gelqf.

Syntax

Fortran:

```
call psormlq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, work, lwork,
info)
```

```
call pdormlq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, work, lwork,
info)
```

C:

```
void psormlq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *desc , float *work , MKL_INT *lwork , MKL_INT *info );
void pdormlq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *desc , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?ormlq routine overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q^* \text{sub}(C)$
<i>trans</i> = 'T':	$Q^T * \text{sub}(C)$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(k) \dots H(2) H(1)$$

as returned by p?gelqf. Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) CHARACTER
= 'L':	Q or Q^T is applied from the left.
= 'R':	Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER
= 'N',	no transpose, Q is applied.
= 'T',	transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).

<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(<i>C</i>) ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) REAL for psormlq DOUBLE PRECISION for pdormlq. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+m-1)</i>), if <i>side</i> = 'L' and (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>), if <i>side</i> = 'R'. The <i>i</i> -th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the <i>k</i> rows of its distributed matrix argument <i>A</i> (<i>ia</i> : <i>ia+k-1</i> , <i>ja</i> :*). <i>A</i> (<i>ia</i> : <i>ia+k-1</i> , <i>ja</i> :*) is modified by the routine but restored on exit.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for psormlq DOUBLE PRECISION for pdormlq Array, size <i>LOCc(ja+k-1)</i> . Contains the scalar factor <i>tau</i> (<i>i</i>) of elementary reflectors <i>H</i> (<i>i</i>) as returned by p?gelqf. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) REAL for psormlq DOUBLE PRECISION for pdormlq Pointer into the local memory to an array of local dimension (<i>lld_c</i> , <i>LOCc(jc+n-1)</i>). Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.
<i>ic</i> , <i>jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descC</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) REAL for psormlq

DOUBLE PRECISION for pdormlq.

Workspace array of dimension of *lwork*.

lwork

(local or global) INTEGER, dimension of the array *work*; must be at least:

If *side* = 'L',

```
lwork ≥ max((mb_a * (mb_a-1))/2, (mpc0+max mqa0) +
numroc(numroc(m + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0,
lcmp), nqc0)) * mb_a + mb_a * mb_a
```

else if *side* = 'R',

```
lwork ≥ max((mb_a * (mb_a-1))/2, (mpc0+nqc0) * mb_a + mb_a * mb_a
```

end if

where

lcm = *lcm*/NPROW with *lcm* = ilcm (NPROW, NPCOL),

iroffa = mod(*ia*-1, *mb_a*),

icoffa = mod(*ja*-1, *nb_a*),

iacol = indxg2p(*ja*, *nb_a*, MYCOL, csrc_a, NPCOL),

mqa0 = numroc(*m+icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),

iroffc = mod(*ic*-1, *mb_c*),

icoffc = mod(*jc*-1, *nb_c*),

icrow = indxg2p(*ic*, *mb_c*, MYROW, rsrc_c, NPROW),

iccol = indxg2p(*jc*, *nb_c*, MYCOL, csrc_c, NPCOL),

mpc0 = numroc(*m+iroffc*, *mb_c*, MYROW, *icrow*, NPROW),

nqc0 = numroc(*n+icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c

Overwritten by the product *Q**sub(*C*), or *Q'* *sub (*C*), or sub(*C*)**Q'*, or sub(*C*)**Q*

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0 : if the i -th argument is an array and the j -entry had an illegal value, then $\text{info} = - (i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.

p?unmlq

Multiples a general matrix by the unitary matrix Q of the LQ factorization formed by p?gelqf.

Syntax

Fortran:

```
call pcunmlq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunmlq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

C:

```
void pcunmlq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );
void pzunmlq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This routine overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C \ (ic:ic+m-1, jc:jc+n-1)$ with

	$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q^* \text{sub}(C)$	$\text{sub}(C)^* Q$
$trans = 'T':$	$Q^H \text{sub}(C)$	$\text{sub}(C)^* Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(k)' \dots H(2)' H(1)'$$

as returned by p?gelqf. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

$side$	(global) CHARACTER
$= 'L':$	Q or Q^H is applied from the left.
$= 'R':$	Q or Q^H is applied from the right.
$trans$	(global) CHARACTER
$= 'N'$,	no transpose, Q is applied.

$= 'C'$, conjugate transpose, Q^H is applied.

m (global) INTEGER. The number of rows in the distributed matrix sub(*C*) ($m \geq 0$).

n (global) INTEGER. The number of columns in the distributed matrix sub(*C*) ($n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix *Q*. Constraints:

If *side* = 'L', $m \geq k \geq 0$

If *side* = 'R', $n \geq k \geq 0$.

a (local)

COMPLEX for pcunm_{lq}

DOUBLE COMPLEX for pzunm_{lq}.

Pointer into the local memory to an array of dimension (*lld_a*, *LOCc(ja+m-1)*), if *side* = 'L', and (*lld_a*, *LOCc(ja+n-1)*), if *side* = 'R', where *lld_a* $\geq \max(1, LOC_r(ia+k-1))$. The *i*-th column must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the *k* rows of its distributed matrix argument *A(ia:ia+k-1, ja:*)*. *A(ia:ia+k-1, ja:*)* is modified by the routine but restored on exit.

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)

COMPLEX for pcunm_{lq}

DOUBLE COMPLEX for pzunm_{lq}

Array, size *LOCc(ia+k-1)*.

Contains the scalar factor *tau* (*i*) of elementary reflectors $H(i)$ as returned by p?gelqf. *tau* is tied to the distributed matrix *A*.

c (local)

COMPLEX for pcunm_{lq}

DOUBLE COMPLEX for pzunm_{lq}.

Pointer into the local memory to an array of local dimension (*lld_c*, *LOCc(jc+n-1)*).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descC (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)

COMPLEX for `pcunmlq`
DOUBLE COMPLEX for `pzunmlq`.

Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of the array *work*; must be at least:
If *side* = 'L',

$$lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + max mqa0) + numroc(numroc(m + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0) * mb_a + mb_a * mb_a$$
else if *side* = 'R',

$$lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + nqc0) * mb_a + mb_a * mb_a$$
end if
where
lcm = *lcm*/NPROW with *lcm* = ilcm (NPROW, NPCOL),
iroffa = mod(*ia*-1, *mb_a*),
icoffa = mod(*ja*-1, *nb_a*),
iacol = indxg2p(*ja*, *nb_a*, MYCOL, *csrc_a*, NPCOL),
mqa0 = numroc(m + *icoffa*, *nb_a*, MYCOL, *iacol*, NPCOL),
iroffc = mod(*ic*-1, *mb_c*),
icoffc = mod(*jc*-1, *nb_c*),
icrow = indxg2p(*ic*, *mb_c*, MYROW, *rsrc_c*, NPROW),
iccol = indxg2p(*jc*, *nb_c*, MYCOL, *csrc_c*, NPCOL),
mpc0 = numroc(m + *iroffc*, *mb_c*, MYROW, *icrow*, NPROW),
nqc0 = numroc(n + *icoffc*, *nb_c*, MYCOL, *iccol*, NPCOL),
ilcm, `indxg2p` and `numroc` are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine `blacs_gridinfo`.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

c Overwritten by the product *Q**sub(*C*), or *Q'**sub (*C*), or sub(*C*)**Q'*, or sub(*C*)**Q*

<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	<p>(global) INTEGER.</p> <p>= 0: the execution is successful.</p> <p>< 0: if the <i>i</i>-th argument is an array and the <i>j</i>-entry had an illegal value, then <i>info</i> = - (<i>i</i>* 100+<i>j</i>), if the <i>i</i>-th argument is a scalar and had an illegal value, then <i>info</i> = -<i>i</i>.</p>

p?geqlf

Computes the QL factorization of a general matrix.

Syntax

Fortran:

```
call psgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqlf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psgeqlf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdgeqlf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcgeqlf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );
void pzgeqlf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?geqlf routine forms the *QL* factorization of a real/complex distributed *m*-by-*n* matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q \cdot L$.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$; (<i>m</i> \geq 0).
<i>n</i>	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ (<i>n</i> \geq 0).
<i>a</i>	<p>(local)</p> <p>REAL for psgeqlf</p>

DOUBLE PRECISION for pdgeqlf

COMPLEX for pcgeqlf

DOUBLE COMPLEX for pzgeqlf

Pointer into the local memory to an array of local dimension ($l1d_a$, $LOCc(ja+n-1)$). Contains the local pieces of the distributed matrix sub(A) to be factored.

ia, ja

(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ia:ia+n-1)$, respectively.

$desca$

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .

$work$

(local)

REAL for psgeqlf

DOUBLE PRECISION for pdgeqlf

COMPLEX for pcgeqlf

DOUBLE COMPLEX for pzgeqlf

Workspace array of dimension $lwork$.

$lwork$

(local or global) INTEGER, dimension of $work$, must be at least $lwork \geq nb_a * (mp0 + nq0 + nb_a)$, where

$iroff = \text{mod}(ia-1, mb_a),$

$icoff = \text{mod}(ja-1, nb_a),$

$iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}),$

$iacol = \text{indxg2p}(ja, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL}),$

$mp0 = \text{numroc}(m+iroff, mb_a, \text{MYROW}, iarow, \text{NPROW}),$

$nq0 = \text{numroc}(n+icoff, nb_a, \text{MYCOL}, iacol, \text{NPCOL})$

`numroc` and `indxg2p` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pixerbla`.

Output Parameters

a

On exit, if $m \geq n$, the lower triangle of the distributed submatrix $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the n -by- n lower triangular matrix L ; if $m \leq n$, the elements on and below the $(n-m)$ -th superdiagonal contain the m -by- n lower trapezoidal matrix L ; the remaining elements, with the array τ , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

τ

(local)

REAL for psgeqlf
 DOUBLE PRECISION for pdgeqlf
 COMPLEX for pcgeqlf
 DOUBLE COMPLEX for pzgeqlf
 Array, size $\text{LOCc}(ja+n-1)$.

Contains the scalar factors of elementary reflectors. τ_u is tied to the distributed matrix A .

work(1) On exit, *work(1)* contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
 = 0: the execution is successful.
 < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja+k-1) * \dots * H(ja+1) * H(ja)$$

where $k = \min(m, n)$

Each $H(i)$ has the form

$$H(i) = I - \tau_u v v'$$

where τ_u is a real/complex scalar, and v is a real/complex vector with $v(m-k+i+1:m) = 0$ and $v(m-k+i) = 1$; $v(1:m-k+i-1)$ is stored on exit in $A(ia:ia+m-k+i-2, ja+n-k+i-1)$, and τ_u in $\tau_u(ja+n-k+i-1)$.

p?orgql

Generates the orthogonal matrix Q of the QL factorization formed by p?geqlf.

Syntax

Fortran:

```
call psorgql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psorgql (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdorgql (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?orgql` routine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = H(k) * \dots * H(2) * H(1)$$

as returned by `p?geqlf`.

Input Parameters

<code>m</code>	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$, ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$, ($m \geq n \geq 0$).
<code>k</code>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
<code>a</code>	(local) REAL for <code>psorgql</code> DOUBLE PRECISION for <code>pdorgql</code> Pointer into the local memory to an array of local dimension (<code>lld_a</code> , <code>LOCc(ja+n-1)</code>). On entry, the j -th column must contain the vector which defines the elementary reflector $H(j)$, $ja+n-k \leq j \leq ja+n-1$, as returned by <code>p?geqlf</code> in the k columns of its distributed matrix argument $A(ia:*, ja+n-k:ja+n-1)$.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<code>desca</code>	(global and local) INTEGER array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix A .
<code>tau</code>	(local) REAL for <code>psorgql</code> DOUBLE PRECISION for <code>pdorgql</code> Array, size <code>LOCc(ja+n-1)</code> . Contains the scalar factors $\tau(j)$ of elementary reflectors $H(j)$. τ is tied to the distributed matrix A .
<code>work</code>	(local) REAL for <code>psorgql</code> DOUBLE PRECISION for <code>pdorgql</code> Workspace array of dimension <code>lwork</code> .
<code>lwork</code>	(local or global) INTEGER, dimension of <code>work</code> , must be at least $lwork \geq nb_a * (nqa0 + mpa0 + nb_a)$, where $iroffa = \text{mod}(ia-1, mb_a),$

```

        icooffa = mod(ja-1, nb_a),
        iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
        iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
        mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
        nqa0 = numroc(n+icooffa, nb_a, MYCOL, iacol, NPCOL)

```

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

<code>a</code>	Contains the local pieces of the m -by- n distributed matrix Q to be factored.
<code>work(1)</code>	On exit, <code>work(1)</code> contains the minimum value of <code>lwork</code> required for optimum performance.
<code>info</code>	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then <code>info = -(i*100+j)</code> , if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

p?ungql

Generates the unitary matrix Q of the QL factorization formed by `p?geqlf`.

Syntax

Fortran:

```
call pcungql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungql(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void pcunmql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );
void pzunmql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: `mkl_scalapack.h`

Description

This routine generates the whole or part of m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first n columns of a product of k elementary reflectors of order m

$Q = (H(k))^H \dots * (H(2))^H * (H(1))^H$ as returned by [p?geqlf](#).

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix $\text{sub}(Q)$ ($m \geq 0$).
n	(global) INTEGER. The number of columns in the submatrix $\text{sub}(Q)$ ($m \geq n \geq 0$).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($n \geq k \geq 0$).
a	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Pointer into the local memory to an array of local dimension ($l1d_a$, $LOCc(ja+n-1)$). On entry, the j -th column must contain the vector which defines the elementary reflector $H(j)$, $ja+n-k \leq j \leq ja+n-1$, as returned by p?geqlf in the k columns of its distributed matrix argument $A(ia:*, ja+n-k: ja+n-1)$.
ia , ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
$desca$	(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .
tau	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Array, size $LOCr(ia+n-1)$. Contains the scalar factors τ_j of elementary reflectors $H(j)$. τ_j is tied to the distributed matrix A .
$work$	(local) COMPLEX for pcungql DOUBLE COMPLEX for pzungql Workspace array of dimension $lwork$.
$lwork$	(local or global) INTEGER, dimension of $work$, must be at least $lwork \geq nb_a * (nqa0 + mpa0 + nb_a)$, where $iroffa = \text{mod}(ia-1, mb_a),$ $icoffa = \text{mod}(ja-1, nb_a),$ $iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}),$

```

iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL)
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW
and NPCOL can be determined by calling the subroutine blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

<i>a</i>	Contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix <i>Q</i> to be factored.
<i>work</i> (1)	On exit, <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (i * 100 + j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?ormql

Multiples a general matrix by the orthogonal matrix Q of the QL factorization formed by p?geqlf.

Syntax

Fortran:

```
call psormql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pdormql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

C:

```

void psormql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );
void pdormql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The p?ormql routine overwrites the general real *m*-by-*n* distributed matrix $\text{sub}(C) = C \ (ic:ic+m-1, jc:jc+n-1)$ with

<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	<i>sub(C)*Q</i>
<i>trans</i> = 'T':	<i>sub(C)*Q^T</i>

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(k)' \dots H(2)' H(1)'$$

as returned by [p?geqlf](#). Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) CHARACTER
	= 'L': Q or Q^T is applied from the left.
	= 'R': Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER
	= 'N', no transpose, Q is applied.
	= 'T', transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix <i>sub(C)</i> , ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix <i>sub(C)</i> , ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) REAL for psormql DOUBLE PRECISION for pdormql . Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+k-1)</i>). The j -th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?gelqf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1).A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If <i>side</i> = 'L', <i>lld_a</i> $\geq \max(1, LOCr(ia+m-1))$, If <i>side</i> = 'R', <i>lld_a</i> $\geq \max(1, LOCr(ia+n-1))$.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local)

```

REAL for psormql
DOUBLE PRECISION for pdormql.

Array, size  $LOC_C(ja+n-1)$ .
Contains the scalar factor  $\tau_u(j)$  of elementary reflectors  $H(j)$  as returned
by p?geqlf.  $\tau_u$  is tied to the distributed matrix  $A$ .
```

c

(local)

```

REAL for psormql
DOUBLE PRECISION for pdormql.

Pointer into the local memory to an array of local dimension ( $lloc_c$ ,
 $LOC_C(jc+n-1)$ ) .
Contains the local pieces of the distributed matrix sub( $C$ ) to be factored.
```

ic, jc

(global) INTEGER. The row and column indices in the global array c
indicating the first row and the first column of the submatrix C ,
respectively.

descC

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor
for the distributed matrix C .

work

(local)

```

REAL for psormql
DOUBLE PRECISION for pdormql.

Workspace array of dimension  $lwork$ .
```

lwork

(local or global) INTEGER, dimension of *work*, must be at least:

```

If side = 'L',
    lwork  $\geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a + nb_a * nb_a)$ 
else if side = 'R',
    lwork  $\geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max npa0) +
    \text{numroc}(\text{numroc}(n + \text{icooffc}, nb_a, 0, 0, \text{NPCOL}), nb_a, 0, 0,
    lcmq), mpc0) * nb_a + nb_a * nb_a$ 
end if
where
lcm = lcm/NPCOL with lcm = ilcm (NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icooffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
npa0 = numroc(n + iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(ic-1, mb_c),
icooffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
```

$mpc0 = \text{numroc}(m+iroffc, mb_c, \text{MYROW}, icrow, \text{NPROW}),$
 $nqc0 = \text{numroc}(n+icoffc, nb_c, \text{MYCOL}, iccol, \text{NPCOL}),$
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pixerbla`.

Output Parameters

c	Overwritten by the product $Q^* \text{sub}(C)$, or $Q'^*\text{sub}(C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$
$work(1)$	On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	(global) INTEGER. = 0: the execution is successful. < 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?unmql

Multiples a general matrix by the unitary matrix Q of the QL factorization formed by p?geqlf.

Syntax

Fortran:

```
call pcunmql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pzunmql(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

C:

```
void pcunmql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunmql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This routine overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C \ (ic:ic+m-1, jc:jc+n-1)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q^* \text{sub}(C)$	$\text{sub}(C)^* Q$
<i>trans</i> = 'C':	$Q^H \text{sub}(C)$	$\text{sub}(C)^* Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$$Q = H(k)' \dots H(2)' H(1)'$$

as returned by [p?geqlf](#). Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) CHARACTER
= 'L':	Q or Q^H is applied from the left.
= 'R':	Q or Q^H is applied from the right.
<i>trans</i>	(global) CHARACTER
= 'N',	no transpose, Q is applied.
= 'C',	conjugate transpose, Q^H is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>a</i>	(local) COMPLEX for pcunmql DOUBLE COMPLEX for pzunmql . Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+k-1)</i>). The <i>j</i> -th column must contain the vector which defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqlf in the <i>k</i> columns of its distributed matrix argument $A(ia:*, ja:ja+k-1).A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If <i>side</i> = 'L', <i>lld_a</i> $\geq \max(1, LOCr(ia+m-1))$, If <i>side</i> = 'R', <i>lld_a</i> $\geq \max(1, LOCr(ia+n-1))$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.

<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for <code>pcunmql</code> DOUBLE COMPLEX for <code>pzunmql</code> Array, size $LOC_C(ia+n-1)$. Contains the scalar factor <i>tau</i> (<i>j</i>) of elementary reflectors $H(j)$ as returned by <code>p?geqlf</code> . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) COMPLEX for <code>pcunmql</code> DOUBLE COMPLEX for <code>pzunmql</code> . Pointer into the local memory to an array of local dimension ($l1d_c$, $LOC_C(jc+n-1)$). Contains the local pieces of the distributed matrix <i>sub(C)</i> to be factored.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descC</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) COMPLEX for <code>pcunmql</code> DOUBLE COMPLEX for <code>pzunmql</code> . Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: <code>If side = 'L',</code> <code>lwork ≥ max((nb_a * (nb_a-1))/2, (nqc0+mpc0)*nb_a + nb_a*nb_a)</code> <code>else if side = 'R',</code> <code>lwork ≥ max((nb_a * (nb_a-1))/2, (nqc0+max npa0) +</code> <code>numroc(numroc(n+icooffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0,</code> <code>lcmq), mpc0)*nb_a) + nb_a*nb_a</code> <code>end if</code> <code>where</code> <code>lcmq = lcm/NPCOL with lcm = ilcm (NPROW, NPCOL),</code> <code>iroffa = mod(ia-1, mb_a),</code> <code>icooffa = mod(ja-1, nb_a),</code> <code>iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),</code> <code>npa0 = numroc (n + iroffa, mb_a, MYROW, iarow, NPROW),</code> <code>iroffc = mod(ic-1, mb_c),</code>

```

icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+icoffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),
ilcm, indxg2p and numroc are ScalAPACK tool functions; MYROW, MYCOL,
NPROW and NPCOL can be determined by calling the subroutine
blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

<i>c</i>	Overwritten by the product $Q^* \text{ sub}(C)$, or $Q' \text{ sub } (C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (i* 100+j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?gerqf

Computes the RQ factorization of a general rectangular matrix.

Syntax

Fortran:

```

call psgerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgerqf(m, n, a, ia, ja, desca, tau, work, lwork, info)

```

C:

```

void psgerqf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdgerqf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcgerqf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );

```

```
void pzgerqf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?gerqf routine forms the QR factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ as

$$A = R \cdot Q$$

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the distributed submatrix $\text{sub}(A)$; $(m \geq 0)$.
<i>n</i>	(global) INTEGER. The number of columns in the distributed submatrix $\text{sub}(A)$; $(n \geq 0)$.
<i>a</i>	(local) REAL for psgeqr DOUBLE PRECISION for pdgeqr COMPLEX for pcgeqr DOUBLE COMPLEX for pzgeqr. Pointer into the local memory to an array of local dimension (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). Contains the local pieces of the distributed matrix $\text{sub}(A)$ to be factored.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i>
<i>work</i>	(local). REAL for psgeqr DOUBLE PRECISION for pdgeqr. COMPLEX for pcgeqr. DOUBLE COMPLEX for pzgeqr Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq mb_a * (mp0 + nq0 + mb_a)$, where $iwoff = \text{mod}(ia-1, mb_a),$ $icoff = \text{mod}(ja-1, nb_a),$

```

        iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
        iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
        mp0 = numroc(m+iroff, mb_a, MYROW, iarow, NPROW),
        nq0 = numroc(n+icoff, nb_a, MYCOL, iacol, NPCOL) and numroc,
indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL
can be determined by calling the subroutine blacs_gridinfo.

```

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a	On exit, if $m \leq n$, the upper triangle of $A(ia:ia+m-1, ja:ja+n-1)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m - n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array τ , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
τ	(local) REAL for psgeqrf DOUBLE PRECISION for pdgeqrf COMPLEX for pcgeqrf DOUBLE COMPLEX for pzgeqrf . Array, size $LOCr(ia+m-1)$. Contains the scalar factor τ of elementary reflectors. τ is tied to the distributed matrix A .
$work(1)$	On exit, $work(1)$ contains the minimum value of $lwork$ required for optimum performance.
$info$	(global) INTEGER. = 0, the execution is successful. < 0, if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ia) * H(ia+1) * \dots * H(ia+k-1),$$

where $k = \min(m, n)$.

Each $H(i)$ has the form

$$H(i) = I - \tau u v^T$$

where τ_{ii} is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and τ_{ii} in $\tau_{ii}(ia+m-k+i-1)$.

p?orgqr

Generates the orthogonal matrix Q of the RQ factorization formed by p?gerqf.

Syntax

Fortran:

```
call psorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgqr(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psorgqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdorgqr (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?orgqr routine generates the whole or part of m -by- n real distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the last m rows of a product of k elementary reflectors of order m

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by p?gerqf.

Input Parameters

m (global) INTEGER. The number of rows in the submatrix sub(Q), ($m \geq 0$).

n (global) INTEGER. The number of columns in the submatrix sub(Q), ($n \geq m \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).

a (local)

REAL for psorgqr

DOUBLE PRECISION for pdorgqr

Pointer into the local memory to an array of local dimension (lld_a , $LOCc(ja+n-1)$). The i -th column must contain the vector which defines the elementary reflector $H(i)$, $ja \leq j \leq ja+k-1$, as returned by p?gerqf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$.

ia, ja

(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau (local)
 REAL for psorgqr
 DOUBLE PRECISION for pdorgqr
 Array, size $LOCc(ja+k-1)$.
 Contains the scalar factor *tau* (*i*) of elementary reflectors $H(i)$ as returned by p?gerqf. *tau* is tied to the distributed matrix *A*.

work (local)
 REAL for psorgqr
 DOUBLE PRECISION for pdorgqr
 Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least $lwork \geq mb_a * (mpa0 + nqa0 + mb_a)$, where
 $iroffa = \text{mod}(ia-1, mb_a),$
 $icoffa = \text{mod}(ja-1, nb_a),$
 $iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),$
 $iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),$
 $mpa0 = \text{numroc}(m+iroffa, mb_a, MYROW, iarow, NPROW),$
 $nqa0 = \text{numroc}(n+icoffa, nb_a, MYCOL, iacol, NPCOL)$
indxg2p and *numroc* are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine *blacs_gridinfo*.
 If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a Contains the local pieces of the *m*-by-*n* distributed matrix Q.

work(1) On exit, *work(1)* contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.
 = 0: the execution is successful.
 < 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?ungrq

Generates the unitary matrix Q of the RQ factorization formed by [p?gerqf](#).

Syntax**Fortran:**

```
call pcungrq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungrq(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void pcungrq (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
void pzungrq (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This routine generates the m -by- n complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

$Q = (H(1))^H * (H(2))^H * \dots * (H(k))^H$ as returned by [p?gerqf](#).

Input Parameters

m	(global) INTEGER. The number of rows in the submatrix sub(Q); ($m \geq 0$).
n	(global) INTEGER. The number of columns in the submatrix sub(Q) ($n \geq m \geq 0$).
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q ($m \geq k \geq 0$).
a	(local) COMPLEX for pcungrq DOUBLE COMPLEX for pzungrqc Pointer into the local memory to an array of dimension (lld_a, LOCc(ja + n - 1)). The i -th row must contain the vector which defines the elementary reflector $H(i)$, $ia + m - k \leq i \leq ia + m - 1$, as returned by p?gerqf in the k rows of its distributed matrix argument $A(ia + m - k : ia + m - 1, ja : *)$.
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) INTEGER array, dimension (dlen_). The array descriptor for the distributed matrix A .

<i>tau</i>	(local) COMPLEX for pcungrq DOUBLE COMPLEX for pzungrq Array, size $LOC_r(ia+m-1)$. Contains the scalar factor <i>tau</i> (<i>i</i>) of elementary reflectors $H(i)$ as returned by p?gerqf. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	(local) COMPLEX for pcungrq DOUBLE COMPLEX for pzungrq Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least $lwork \geq mb_a * (mpa0 + nqa0 + mb_a)$, where <i>iroffa</i> = mod(<i>ia</i> -1, <i>mb_a</i>), <i>icoffa</i> = mod(<i>ja</i> -1, <i>nb_a</i>), <i>iarow</i> = indxg2p(<i>ia</i> , <i>mb_a</i> , MYROW, rsrc_a, NPROW), <i>iacol</i> = indxg2p(<i>ja</i> , <i>nb_a</i> , MYCOL, csrc_a, NPCOL), <i>mpa0</i> = numroc(<i>m+iroffa</i> , <i>mb_a</i> , MYROW, <i>iarow</i> , NPROW), <i>nqa0</i> = numroc(<i>n+icoffa</i> , <i>nb_a</i> , MYCOL, <i>iacol</i> , NPCOL) indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo. If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

<i>a</i>	Contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix <i>Q</i> .
<i>work</i> (1)	On exit <i>work</i> (1) contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?ormrq

Multiples a general matrix by the orthogonal matrix *Q* of the RQ factorization formed by p?gerqf.

Syntax

Fortran:

```
call psormrq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pdormrq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

C:

```
void psormrq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );
void pdormrq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?ormrq` routine overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

$side = 'L'$	$side = 'R'$
$trans = 'N':$	$Q * \text{sub}(C)$
$trans = 'T':$	$Q^T * \text{sub}(C)$
	$\text{sub}(C) * Q$
	$\text{sub}(C) * Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1) \ H(2) \dots \ H(k)$$

as returned by `p?gerqf`. Q is of order m if $side = 'L'$ and of order n if $side = 'R'$.

Input Parameters

<i>side</i>	(global) CHARACTER
= 'L':	Q or Q^T is applied from the left.
= 'R':	Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER
= 'N',	no transpose, Q is applied.
= 'T',	transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If $side = 'L'$, $m \geq k \geq 0$ If $side = 'R'$, $n \geq k \geq 0$.

<i>a</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+m-1)</i>) if <i>side</i> = 'L', and (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>) if <i>side</i> = 'R'. The <i>i</i> -th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gerqf in the <i>k</i> rows of its distributed matrix argument $A(ia:ia+k-1, ja:*) . A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for psormqr DOUBLE PRECISION for pdormqr Array, size <i>LOCc(ja+k-1)</i> . Contains the scalar factor <i>tau</i> (<i>i</i>) of elementary reflectors $H(i)$ as returned by p?gerqf. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) REAL for psormrq DOUBLE PRECISION for pdormrq Pointer into the local memory to an array of local dimension (<i>lld_c</i> , <i>LOCc(jc+n-1)</i>). Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descC</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) REAL for psormrq DOUBLE PRECISION for pdormrq. Workspace array of dimension <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER, dimension of <i>work</i> , must be at least: If <i>side</i> = 'L',

```

lwork ≥ max((mb_a * (mb_a-1))/2, (mpc0 + max(mqa0 +
numroc(numroc(n+iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0,
lcmp), nqc0)) * mb_a) + mb_a * mb_a

else if side = 'R',
lwork ≥ max((mb_a * (mb_a-1))/2, (mpc0 + nqc0) * mb_a) +
mb_a * mb_a

end if

where

lcm = lcm/NPROW with lcm = ilcm (NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icooffa = mod(ja-1, nb_a),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(n+icooffa, nb_a, MYCOL, iacol, NPCOL),
iroffc = mod(ic-1, mb_c),
icooffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icooffc, nb_c, MYCOL, iccol, NPCOL),
ilcm, indxg2p and numroc are ScalAPACK tool functions; MYROW, MYCOL,
NPROW and NPCOL can be determined by calling the subroutine
blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

<i>c</i>	Overwritten by the product $Q^* \text{sub}(C)$, or $Q'^*\text{sub}(C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (i * 100+j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?unmrq

Multiples a general matrix by the unitary matrix Q of the RQ factorization formed by p?gerqf.

Syntax

Fortran:

```
call pcunmrq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunmrq(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

C:

```
void pcunmrq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );
```

```
void pzunmrq (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This routine overwrites the general complex m -by- n distributed matrix sub (C) = $C(ic:ic+m-1, jc:jc+n-1)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q^* \text{sub}(C)$	$\text{sub}(C)*Q$
<i>trans</i> = 'C':	$Q^H * \text{sub}(C)$	$\text{sub}(C)*Q^H$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(1)' H(2)' \dots H(k)'$

as returned by [p?gerqf](#). Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) CHARACTER ='L': Q or Q^H is applied from the left. ='R': Q or Q^H is applied from the right.
<i>trans</i>	(global) CHARACTER ='N', no transpose, Q is applied. ='C', conjugate transpose, Q^H is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$, ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$, ($n \geq 0$).

k (global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints:

If $\text{side} = \text{'L'}$, $m \geq k \geq 0$

If $\text{side} = \text{'R'}$, $n \geq k \geq 0$.

a (local)

COMPLEX for `pcunmrq`

DOUBLE COMPLEX for `pzunmrq`.

Pointer into the local memory to an array of dimension (lld_a , $\text{LOCc}(ja + m - 1)$) if $\text{side} = \text{'L'}$, and (lld_a , $\text{LOCc}(ja + n - 1)$) if $\text{side} = \text{'R'}$. The i -th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia + k - 1$, as returned by `p?gerqf` in the k rows of its distributed matrix argument $A(ia:ia + k - 1, ja^*) \cdot A(ia:ia + k - 1, ja^*)$ is modified by the routine but restored on exit.

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

descA (global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix *A*.

tau (local)

COMPLEX for `pcunmrq`

DOUBLE COMPLEX for `pzunmrq`

Array, size $\text{LOCc}(ja + k - 1)$.

Contains the scalar factor *tau* (i) of elementary reflectors $H(i)$ as returned by `p?gerqf`. *tau* is tied to the distributed matrix *A*.

c (local)

COMPLEX for `pcunmrq`

DOUBLE COMPLEX for `pzunmrq`.

Pointer into the local memory to an array of local dimension (lld_c , $\text{LOCc}(jc + n - 1)$).

Contains the local pieces of the distributed matrix $\text{sub}(C)$ to be factored.

ic, jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descC (global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix *C*.

work (local)

COMPLEX for `pcunmrq`

DOUBLE COMPLEX for `pzunmrq`.

Workspace array of dimension of *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:

```

If side = 'L',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 +
max(mqa0+numroc(numroc(n+iroffc, mb_a, 0, 0, NPROW), mb_a,
0, 0, lcmp), nqc0)*mb_a) + mb_a*mb_a

else if side = 'R',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a) +
mb_a*mb_a

end if

where

lcm = lcm/NPROW with lcm = ilcm(NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(m+icoffa, nb_a, MYCOL, iacol, NPCOL),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.
```

Output Parameters

<i>c</i>	Overwritten by the product $Q^* \text{sub}(C)$ or $Q'^*\text{sub}(C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> * 100+j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?tzrzf

Reduces the upper trapezoidal matrix A to upper triangular form.

Syntax**Fortran:**

```
call pstzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdtzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pctzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pztzrzf(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void pstzrzf (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdtzrzf (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pctzrzf (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );
void pztzrzf (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?tzrzf` routine reduces the m -by- n ($m \leq n$) real/complex upper trapezoidal matrix $\text{sub}(A) = (ia:ia+m-1, ja:ja+n-1)$ to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix A is factored as

$$A = (R \ 0) * Z,$$

where Z is an n -by- n orthogonal/unitary matrix and R is an m -by- m upper triangular matrix.

Input Parameters

<code>m</code>	(global) INTEGER. The number of rows in the submatrix $\text{sub}(A)$; ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns in the submatrix $\text{sub}(A)$ ($n \geq 0$).
<code>a</code>	(local) REAL for <code>pstzrzf</code> DOUBLE PRECISION for <code>pdtzrzf</code> . COMPLEX for <code>pctzrzf</code> . DOUBLE COMPLEX for <code>pztzrzf</code> .

Pointer into the local memory to an array of dimension (lld_a , $LOCc(ja + n - 1)$). Contains the local pieces of the m -by- n distributed matrix sub (A) to be factored.

ia, ja
(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

desca
(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .

work
(local)
REAL for `pstzrzf`
DOUBLE PRECISION for `pdtzrzf`.
COMPLEX for `pctzrzf`.
DOUBLE COMPLEX for `pztzrzf`.

Workspace array of dimension $lwork$.

lwork
(local or global) INTEGER, dimension of $work$, must be at least $lwork \geq mb_a * (mp0 + nq0 + mb_a)$, where
 $iroff = \text{mod}(ia-1, mb_a)$,
 $icoff = \text{mod}(ja-1, nb_a)$,
 $iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW})$,
 $iacol = \text{indxg2p}(ja, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL})$,
 $mp0 = \text{numroc}(m+iroff, mb_a, \text{MYROW}, iarow, \text{NPROW})$,
 $nq0 = \text{numroc}(n+icoff, nb_a, \text{MYCOL}, iacol, \text{NPCOL})$
`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.
If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a
On exit, the leading m -by- m upper triangular part of sub(A) contains the upper triangular matrix R , and elements $m+1$ to n of the first m rows of sub (A), with the array τ , represent the orthogonal/unitary matrix Z as a product of m elementary reflectors.

work(1)
On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.

tau
(local)
REAL for `pstzrzf`
DOUBLE PRECISION for `pdtzrzf`.
COMPLEX for `pctzrzf`.

DOUBLE COMPLEX for pztzrzf.

Array, size $\text{LOC}_r(ia+m-1)$.

Contains the scalar factor of elementary reflectors. τ is tied to the distributed matrix A .

info (global) INTEGER.

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -entry had an illegal value, then $\text{info} = -(i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.

Application Notes

The factorization is obtained by the Householder's method. The k -th transformation matrix, $Z(k)$, which is or whose conjugate transpose is used to introduce zeros into the $(m - k + 1)$ -th row of $\text{sub}(A)$, is given in the form

$$\begin{pmatrix} \dots & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & \dots \end{pmatrix} = \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix} \begin{pmatrix} \dots & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & \dots \end{pmatrix}$$

where

$$T(k) = I - \tau u(k) u(k)^T,$$

$$\begin{pmatrix} \dots & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & \dots \end{pmatrix}$$

τ is a scalar and $Z(k)$ is an $(n - m)$ element vector. τ and $Z(k)$ are chosen to annihilate the elements of the k -th row of $\text{sub}(A)$. The scalar τ is returned in the k -th element of τ and the vector $u(k)$ in the k -th row of $\text{sub}(A)$, such that the elements of $Z(k)$ are in $a(k, m + 1), \dots, a(k, n)$. The elements of R are returned in the upper triangular part of $\text{sub}(A)$. Z is given by

$$Z = Z(1) * Z(2) * \dots * Z(m).$$

p?ormrz

Multiples a general matrix by the orthogonal matrix from a reduction to upper triangular form formed by p?tzrzf.

Syntax

Fortran:

```
call psormrz(side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

```
call pdormrz(side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work, lwork, info)
```

C:

```

void psormrz (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_INT *l , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau ,
float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , float *work , MKL_INT *lwork ,
MKL_INT *info );

void pdormrz (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_INT *l , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau ,
double *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , double *work , MKL_INT
*lwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

This routine overwrites the general real m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q^* \text{sub}(C)$
<i>trans</i> = 'T':	$Q^T \text{sub}(C)$
	$\text{sub}(C)^* Q^T$

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

$$Q = H(1) H(2) \dots H(k)$$

as returned by [p?tzrzf](#). Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) CHARACTER
= 'L':	Q or Q^T is applied from the left.
= 'R':	Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER
= 'N',	no transpose, Q is applied.
= 'T',	transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>l</i>	(global) The columns of the distributed submatrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. If <i>side</i> = 'L', $m \geq l \geq 0$

If $\text{side} = \text{'R'}$, $n \geq l \geq 0$.

a (local)

REAL for psormrz

DOUBLE PRECISION for pdormrz.

Pointer into the local memory to an array of dimension (lld_a , $\text{LOCc}(ja+m-1)$) if $\text{side} = \text{'L'}$, and (lld_a , $\text{LOCc}(ja+n-1)$) if $\text{side} = \text{'R'}$, where $\text{lld}_a \geq \max(1, \text{LOCr}(ia+k-1))$.

The i -th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?tzrzf in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*) \cdot A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.

ia, ja

(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

descA

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix *A*.

tau

(local)

REAL for psormrz

DOUBLE PRECISION for pdormrz

Array, size $\text{LOCc}(ia+k-1)$.

Contains the scalar factor *tau* (i) of elementary reflectors $H(i)$ as returned by p?tzrzf. *tau* is tied to the distributed matrix *A*.

c

(local)

REAL for psormrz

DOUBLE PRECISION for pdormrz

Pointer into the local memory to an array of local dimension (lld_c , $\text{LOCc}(jc+n-1)$).

Contains the local pieces of the distributed matrix sub(*C*) to be factored.

ic, jc

(global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descC

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix *C*.

work

(local)

REAL for psormrz

DOUBLE PRECISION for pdormrz.

Workspace array of dimension of *lwork*.

lwork

(local or global) INTEGER, dimension of *work*, must be at least:

```

If side = 'L',
lwork  $\geq \max((mb\_a * (mb\_a - 1)) / 2, (mpc0 + \max(mqa0 +
numroc(numroc(n+iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0,
1cmp), nqc0)) * mb_a) + mb\_a * mb\_a

else if side = 'R',
lwork  $\geq \max((mb\_a * (mb\_a - 1)) / 2, (mpc0 + nqc0) * mb\_a) +
mb\_a * mb\_a

end if

where

lcm = lcm/NPROW with lcm = ilcm (NPROW, NPCOL),
iroffa = mod(ia-1, mb_a), icoffa = mod(ja-1, nb_a),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL,
NPROW and NPCOL can be determined by calling the subroutine
blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is
assumed; the routine only calculates the minimum and optimal size for all
work arrays. Each of these values is returned in the first entry of the
corresponding work array, and no error message is issued by pxerbla.$$ 
```

Output Parameters

<i>c</i>	Overwritten by the product <i>Q</i> *sub(<i>C</i>), or <i>Q</i> '*sub (<i>C</i>), or sub(<i>C</i>)* <i>Q</i> ', or sub(<i>C</i>)* <i>Q</i>
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> * 100+j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?unmrz

Multiples a general matrix by the unitary transformation matrix from a reduction to upper triangular form determined by p?tzrzf.

Syntax**Fortran:**

```
call pcunmrz(side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunmrz(side, trans, m, n, k, l, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

C:

```
void pcunmrz (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_INT *l , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_Complex8 *tau , MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );
void pzunmrz (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_INT *l , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_Complex16 *tau , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This routine overwrites the general complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q^* \text{sub}(C)$
<i>trans</i> = 'C':	$\text{sub}(C)^* Q$

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(1)' H(2)' \dots H(k)'$

as returned by pctzrzf/pztzrzf. Q is of order m if *side* = 'L' and of order n if *side* = 'R'.

Input Parameters

<i>side</i>	(global) CHARACTER
= 'L':	Q or Q^H is applied from the left.
= 'R':	Q or Q^H is applied from the right.
<i>trans</i>	(global) CHARACTER
= 'N',	no transpose, Q is applied.
= 'C',	conjugate transpose, Q^H is applied.

<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub(<i>C</i>), ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub(<i>C</i>), ($n \geq 0$).
<i>k</i>	(global) INTEGER. The number of elementary reflectors whose product defines the matrix <i>Q</i> . Constraints: If <i>side</i> = 'L', $m \geq k \geq 0$ If <i>side</i> = 'R', $n \geq k \geq 0$.
<i>l</i>	(global) INTEGER. The columns of the distributed submatrix sub(<i>A</i>) containing the meaningful part of the Householder reflectors. If <i>side</i> = 'L', $m \geq l \geq 0$ If <i>side</i> = 'R', $n \geq l \geq 0$.
<i>a</i>	(local) COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+m-1)</i>) if <i>side</i> = 'L', and (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>) if <i>side</i> = 'R', where <i>lld_a</i> $\geq \max(1, LOCr(ja+k-1))$. The <i>i</i> -th row must contain the vector which defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gerqf in the <i>k</i> rows of its distributed matrix argument <i>A</i> (<i>ia</i> : <i>ia+k-1</i> , <i>ja</i> *). <i>A</i> (<i>ia</i> : <i>ia+k-1</i> , <i>ja</i> *) is modified by the routine but restored on exit.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz Array, size <i>LOCc(ia+k-1)</i> . Contains the scalar factor <i>tau</i> (<i>i</i>) of elementary reflectors $H(i)$ as returned by p?gerqf . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) COMPLEX for pcunmrz DOUBLE COMPLEX for pzunmrz. Pointer into the local memory to an array of local dimension (<i>lld_c</i> , <i>LOCc(jc+n-1)</i>). Contains the local pieces of the distributed matrix sub(<i>C</i>) to be factored.

ic, jc (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descC (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work (local)
 COMPLEX for `pcunmrz`
 DOUBLE COMPLEX for `pzunmrz`.
 Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:
If *side* = 'L',

$$lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + \max(mqa0 + \text{numroc}(\text{numroc}(n + iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, 1cmp), nqc0)) * mb_a) + mb_a * mb_a$$
else if *side* = 'R',

$$lwork \geq \max((mb_a * (mb_a - 1)) / 2, (mpc0 + nqc0) * mb_a) + mb_a * mb_a$$
end if
where

$$\begin{aligned} 1cm &= lcm/NPROW \text{ with } lcm = \text{ilcm}(NPROW, NPCOL), \\ iroffa &= \text{mod}(ia-1, mb_a), \\ icooffa &= \text{mod}(ja-1, nb_a), \\ iacol &= \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL), \\ mqa0 &= \text{numroc}(m + icooffa, nb_a, MYCOL, iacol, NPCOL), \\ iroffc &= \text{mod}(ic-1, mb_c), \\ icooffc &= \text{mod}(jc-1, nb_c), \\ icrow &= \text{indxg2p}(ic, mb_c, MYROW, rsrc_c, NPROW), \\ iccol &= \text{indxg2p}(jc, nb_c, MYCOL, csrc_c, NPCOL), \\ mpc0 &= \text{numroc}(m + iroffc, mb_c, MYROW, icrow, NPROW), \\ nqc0 &= \text{numroc}(n + icooffc, nb_c, MYCOL, iccol, NPCOL), \end{aligned}$$
 ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

<i>c</i>	Overwritten by the product $Q^* \text{sub}(C)$, or $Q'^* \text{sub}(C)$, or $\text{sub}(C)*Q'$, or $\text{sub}(C)*Q$
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?ggqrft

Computes the generalized QR factorization.

Syntax

Fortran:

```
call psggqrft(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork,
info)

call pdggqrft(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork,
info)

call pcggqrft(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork,
info)

call pzggqrft(n, m, p, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork,
info)
```

C:

```
void psggqrft (MKL_INT *n , MKL_INT *m , MKL_INT *p , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *taua , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , float *taub , float *work , MKL_INT *lwork , MKL_INT *info );

void pdggqrft (MKL_INT *n , MKL_INT *m , MKL_INT *p , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *taua , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , double *taub , double *work , MKL_INT *lwork , MKL_INT *info );

void pcggqrft (MKL_INT *n , MKL_INT *m , MKL_INT *p , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *taua , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *taub , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzggqrft (MKL_INT *n , MKL_INT *m , MKL_INT *p , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *taua , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *taub , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?ggqrft routine forms the generalized QR factorization of an *n*-by-*m* matrix

`sub(A) = A(ia:ia+n-1, ja:ja+m-1)`

and an n -by- p matrix

`sub(B) = B(ib:ib+n-1, jb:jb+p-1):`

as

`sub(A) = Q*R, sub(B) = Q*T*Z,`

where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

If $n \geq m$

$$R = \begin{pmatrix} R_{11} \\ 0 \end{pmatrix} \begin{matrix} m \\ n-m \\ m \end{matrix}$$

or if $n < m$

$$R = \begin{pmatrix} R_{11} & R_{12} \end{pmatrix} \begin{matrix} n \\ n-m \\ n \end{matrix}$$

where R_{11} is upper triangular, and

$$T = \begin{pmatrix} 0 & T_{12} \\ p-n & n \end{pmatrix} n, \text{ if } n \leq p,$$

$$\text{or } T = \begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix} \begin{pmatrix} n-p \\ p \end{pmatrix}, \text{ if } n > p,$$

where T_{12} or T_{21} is an upper triangular matrix.

In particular, if $\text{sub}(B)$ is square and nonsingular, the GQR factorization of $\text{sub}(A)$ and $\text{sub}(B)$ implicitly gives the QR factorization of $\text{inv}(\text{sub}(B))^* \text{sub}(A)$:

`inv(sub(B))*sub(A) = Z^H*(inv(T)*R)`

Input Parameters

`n` (global) INTEGER. The number of rows in the distributed matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).

`m` (global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$ ($m \geq 0$).

`p` INTEGER. The number of columns in the distributed matrix $\text{sub}(B)$ ($p \geq 0$).

`a` (local)

REAL for psggqrf
 DOUBLE PRECISION for pdggqrf
 COMPLEX for pcggqrf
 DOUBLE COMPLEX for pzggqrf.

Pointer into the local memory to an array of dimension (lld_a , $LOCc(ja+m-1)$). Contains the local pieces of the n -by- m matrix sub(A) to be factored.

ia, ja
 (global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

desca
 (global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .

b
 (local)
 REAL for psggqrf
 DOUBLE PRECISION for pdggqrf
 COMPLEX for pcggqrf
 DOUBLE COMPLEX for pzggqrf.

Pointer into the local memory to an array of dimension (lld_b , $LOCc(jb+p-1)$). Contains the local pieces of the n -by- p matrix sub(B) to be factored.

ib, jb
 (global) INTEGER. The row and column indices in the global array b indicating the first row and the first column of the submatrix B , respectively.

descb
 (global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix B .

work
 (local)
 REAL for psggqrf
 DOUBLE PRECISION for pdggqrf
 COMPLEX for pcggqrf
 DOUBLE COMPLEX for pzggqrf.
 Workspace array of dimension $lwork$.

lwork
 (local or global) INTEGER. Dimension of *work*, must be at least

$$lwork \geq \max(nb_a * (npa0 + mqa0 + nb_a), \max((nb_a * (nb_a - 1)) / 2, (pqb0 + npb0) * nb_a) + nb_a * nb_a, mb_b * (npb0 + pqb0 + mb_b)),$$
 where

$$\begin{aligned}iroffa &= \text{mod}(ia-1, mb_A), \\icoffa &= \text{mod}(ja-1, nb_a), \\iarow &= \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}),\end{aligned}$$

```

iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
nra0 = numroc (n+iroffa, mb_a, MYROW, iarow, NPROW),
mqa0 = numroc (m+icooffa, nb_a, MYCOL, iacol, NPCOL)
iroffb = mod(ib-1, mb_b),
icoffb = mod(jb-1, nb_b),
ibrow = indxg2p(ib, mb_b, MYROW, rsrc_b, NPROW),
ibcol = indxg2p(jb, nb_b, MYCOL, csrc_b, NPCOL),
npb0 = numroc (n+iroffa, mb_b, MYROW, Ibrow, NPROW),
pqb0 = numroc (m+icoffb, nb_b, MYCOL, ibcol, NPCOL)
and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW
and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If lwork = -1, then lwork is global input and a workspace query is
assumed; the routine only calculates the minimum and optimal size for all
work arrays. Each of these values is returned in the first entry of the
corresponding work array, and no error message is issued by pxerbla.

```

Output Parameters

a

On exit, the elements on and above the diagonal of sub (*A*) contain the $\min(n, m)$ -by- m upper trapezoidal matrix *R* (*R* is upper triangular if $n \geq m$); the elements below the diagonal, with the array *taua*, represent the orthogonal/unitary matrix *Q* as a product of $\min(n,m)$ elementary reflectors. (See Application Notes below).

taua, *taub*

(local)

REAL for psggqr
 DOUBLE PRECISION for pdggqr
 COMPLEX for pcggqr
 DOUBLE COMPLEX for pzggqr.

Arrays, size $LOCc(ja+\min(n,m)-1)$ for *taua* and $LOCr(ib+n-1)$ for *taub*.

The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Q*. *taua* is tied to the distributed matrix *A*. (See Application Notes below).

The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*. *taub* is tied to the distributed matrix *B*. (See Application Notes below).

work(1)

On exit *work(1)* contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0 : if the i -th argument is an array and the j -entry had an illegal value, then $\text{info} = - (i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja) * H(ja+1) * \dots * H(ja+k-1),$$

where $k = \min(n, m)$.

Each $H(i)$ has the form

$$H(i) = i - \tau_{aua} v v'$$

where τ_{aua} is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:n)$ is stored on exit in $A(ia+i:ia+n-1, ja+i-1)$, and τ_{aua} in $\tau_{aua}(ja+i-1)$. To form Q explicitly, use ScaLAPACK subroutine [p?orgqr/p?ungqr](#). To use Q to update another matrix, use ScaLAPACK subroutine [p?ormqr/p?unmqr](#).

The matrix Z is represented as a product of elementary reflectors

$$Z = H(ib) * H(ib+1) * \dots * H(ib+k-1), \text{ where } k = \min(n, p).$$

Each $H(i)$ has the form

$$H(i) = i - \tau_{aub} v v'$$

where τ_{aub} is a real/complex scalar, and v is a real/complex vector with $v(p-k+i+1:p) = 0$ and $v(p-k+i) = 1$; $v(1:p-k+i-1)$ is stored on exit in $B(ib+n-k+i-1, jb:jb+p-k+i-2)$, and τ_{aub} in $\tau_{aub}(ib+n-k-i-1)$. To form Z explicitly, use ScaLAPACK subroutine [p?orgqr/p?ungqr](#). To use Z to update another matrix, use ScaLAPACK subroutine [p?ormrq/p?unmqr](#).

p?ggrqf

Computes the generalized RQ factorization.

Syntax

Fortran:

```
call psggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
call pdggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
call pcggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
call pzggrqf(m, p, n, a, ia, ja, desca, taua, b, ib, jb, descb, taub, work, lwork, info)
```

C:

```
void psggrqf (MKL_INT *m , MKL_INT *p , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *taua , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , float *taub , float *work , MKL_INT *lwork , MKL_INT *info );
void pdggrqf (MKL_INT *m , MKL_INT *p , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *taua , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , double *taub , double *work , MKL_INT *lwork , MKL_INT *info );
```

```

void pcggrqf (MKL_INT *m , MKL_INT *p , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *taua , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *taub , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzggrqf (MKL_INT *m , MKL_INT *p , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *taua , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *taub , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );

```

Include Files

- C: mkl scalapack.h

Description

The `p?gqrqf` routine forms the generalized RQ factorization of an m -by- n matrix $\text{sub}(A) = (ia:ia+m-1, ja:ja+n-1)$ and a p -by- n matrix $\text{sub}(B) = (ib:ib+p-1, ja:ja+n-1)$:

$$\text{sub}(A) = R^*Q, \quad \text{sub}(B) = Z^*T^*Q,$$

where Q is an n -by- n orthogonal/unitary matrix, Z is a p -by- p orthogonal/unitary matrix, and R and T assume one of the forms:

$$R = m \begin{pmatrix} 0 & R_{12} \end{pmatrix}, \text{ if } m \leq n,$$

or

$$R = \begin{pmatrix} R_{11} \\ R_{12} \\ \vdots \\ R_m \end{pmatrix}^m - n \quad , \text{ if } m > n$$

where R_{11} or R_{21} is upper triangular, and

$$T = \begin{pmatrix} T_{11} \\ 0 \end{pmatrix} \begin{matrix} n \\ p-n \end{matrix}, \text{ if } p \geq n$$

or

10.1007/s00339-017-0363-1

where T^{11} is upper triangular.

In particular, if $\text{sub}(B)$ is square and nonsingular, the GRQ factorization of $\text{sub}(A)$ and $\text{sub}(B)$ implicitly gives the RQ factorization of $\text{sub}(A)^*\text{inv}(\text{sub}(B))$:

$$\text{sub}(A) * \text{inv}(\text{sub}(B)) = (R * \text{inv}(T)) * Z'$$

where $\text{inv}(\text{sub}(B))$ denotes the inverse of the matrix $\text{sub}(B)$, and Z' denotes the transpose (conjugate transpose) of matrix Z .

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows in the distributed matrices sub (<i>A</i>) ($m \geq 0$).
<i>p</i>	INTEGER. The number of rows in the distributed matrix sub(<i>B</i>) ($p \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrices sub(<i>A</i>) and sub(<i>B</i>) ($n \geq 0$).
<i>a</i>	(local) REAL for psggrqf DOUBLE PRECISION for pdggrqf COMPLEX for pcggrqf DOUBLE COMPLEX for pzggrqf. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja</i> + <i>n</i> -1)). Contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix sub(<i>A</i>) to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desc_a</i>	(global and local) INTEGER array, dimension (<i>dlen_a</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) REAL for psggrqf DOUBLE PRECISION for pdggrqf COMPLEX for pcggrqf DOUBLE COMPLEX for pzggrqf. Pointer into the local memory to an array of dimension (<i>lld_b</i> , <i>LOCc(jb</i> + <i>n</i> -1)). Contains the local pieces of the <i>p</i> -by- <i>n</i> matrix sub(<i>B</i>) to be factored.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>b</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>desc_b</i>	(global and local) INTEGER array, dimension (<i>dlen_b</i>). The array descriptor for the distributed matrix <i>B</i> .
<i>work</i>	(local) REAL for psggrqf DOUBLE PRECISION for pdggrqf COMPLEX for pcggrqf DOUBLE COMPLEX for pzggrqf.

Workspace array of dimension of *lwork*.

lwork

(local or global) INTEGER.

Dimension of *work*, must be at least $lwork \geq \max(mb_a * (mpa0 + nqa0 + mb_a), \max((mb_a * (mb_a - 1)) / 2, (ppb0 + nqb0) * mb_a) + mb_a * mb_a, nb_b * (ppb0 + nqb0 + nb_b))$, where

```

iroffa = mod(ia-1, mb_A),
icooffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(ja, nb_a, MYCOL, csrc_a, NPCOL),
mpa0 = numroc (m+iroffa, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc (m+icooffa, nb_a, MYCOL, iacol, NPCOL)
iroffb = mod(ib-1, mb_b),
icoffb = mod(jb-1, nb_b),
ibrow = indxg2p(ib, mb_b, MYROW, rsrc_b, NPROW),
ibcol = indxg2p(jb, nb_b, MYCOL, csrc_b, NPCOL),
ppb0 = numroc (p+iroffb, mb_b, MYROW, ibrow, NPROW),
nqb0 = numroc (n+icoffb, nb_b, MYCOL, ibcol, NPCOL)
and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW
and NPCOL can be determined by calling the subroutine blacs_gridinfo.

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a

On exit, if $m \leq n$, the upper triangle of $A(ia:ia+m-1, ja+n-m:ja+n-1)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m-n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array *taua*, represent the orthogonal/unitary matrix Q as a product of $\min(n, m)$ elementary reflectors (see *Application Notes* below).

taua, *taub*

(local)

REAL for psggqr
DOUBLE PRECISION for pdggqr
COMPLEX for pcggqr
DOUBLE COMPLEX for pzggqr.

Arrays, size $LOCr(ia+m-1)$ for *taua* and $LOCc(jb+\min(p, n)-1)$ for *taub*.

The array *taua* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q . *taua* is tied to the distributed matrix *A*. (See *Application Notes* below).

The array *taub* contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix *Z*. *taub* is tied to the distributed matrix *B*. (See *Application Notes* below).

work(1)

On exit *work*(1) contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100+j), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrix *Q* is represented as a product of elementary reflectors

$$Q = H(ia)*H(ia+1)*\dots*H(ia+k-1),$$

where $k = \min(m, n)$.

Each *H*(*i*) has the form

$$H(i) = i - taua * v * v'$$

where *tau*_a is a real/complex scalar, and *v* is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ is stored on exit in *A*(*ia+m-k+i-1*, *ja:j+n-k+i-2*), and *tau*_a in *tau*_a(*ia+m-k-1*). To form *Q* explicitly, use ScaLAPACK subroutine [p?orgqr/p?ungqr](#). To use *Q* to update another matrix, use ScaLAPACK subroutine [p?ormqr/p?unmqr](#).

The matrix *Z* is represented as a product of elementary reflectors

$$Z = H(jb)*H(jb+1)*\dots*H(jb+k-1), \text{ where } k = \min(p, n).$$

Each *H*(*i*) has the form

$$H(i) = i - taub * v * v'$$

where *taub* is a real/complex scalar, and *v* is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:p)$ is stored on exit in *B*(*ib+i:ib+p-1*, *jb+i-1*), and *taub* in *taub*(*jb+i-1*). To form *Z* explicitly, use ScaLAPACK subroutine [p?orgqr/p?ungqr](#). To use *Z* to update another matrix, use ScaLAPACK subroutine [p?ormqr/p?unmqr](#).

Symmetric Eigenproblems

To solve a symmetric eigenproblem with ScaLAPACK, you usually need to reduce the matrix to real tridiagonal form *T* and then find the eigenvalues and eigenvectors of the tridiagonal matrix *T*. ScaLAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in [Table "Computational Routines for Solving Symmetric Eigenproblems"](#).

There are different routines for symmetric eigenproblems, depending on whether you need eigenvalues only or eigenvectors as well, and on the algorithm used (either the *QTQ* algorithm, or bisection followed by inverse iteration).

Computational Routines for Solving Symmetric Eigenproblems

Operation	Dense symmetric/ Hermitian matrix	Orthogonal/unitary matrix	Symmetric tridiagonal matrix
Reduce to tridiagonal form $A = QTQ^H$	p?sytrd/p?hetrd		
Multiply matrix after reduction		p?ormtr/p?unmtr	
Find all eigenvalues and eigenvectors of a tridiagonal matrix T by a QTQ method			steqr2*
Find selected eigenvalues of a tridiagonal matrix T via bisection			p?stebz
Find selected eigenvectors of a tridiagonal matrix T by inverse iteration			p?stein

* This routine is described as part of auxiliary ScaLAPACK routines.

p?sytrd

Reduces a symmetric matrix to real symmetric
tridiagonal form by an orthogonal similarity
transformation.

Syntax

Fortran:

```
call pssytrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pdsytrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

C:

```
void pssytrd (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *d , float *e , float *tau , float *work , MKL_INT *lwork , MKL_INT
*info );
void pdsytrd (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *d , double *e , double *tau , double *work , MKL_INT *lwork ,
MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?sytrd` routine reduces a real symmetric matrix `sub(A)` to symmetric tridiagonal form T by an orthogonal similarity transformation:

$$Q' * \text{sub}(A) * Q = T,$$

where `sub(A) = A(ia:ia+n-1, ja:ja+n-1)`.

Input Parameters

<code>uplo</code>	(global) CHARACTER.
	Specifies whether the upper or lower triangular part of the symmetric matrix <code>sub(A)</code> is stored:
If <code>uplo = 'U'</code> ,	upper triangular
If <code>uplo = 'L'</code> ,	lower triangular

n (global) INTEGER. The order of the distributed matrix sub(*A*) ($n \geq 0$).

a (local)

REAL for pssytrd

DOUBLE PRECISION for pdsytrd.

Pointer into the local memory to an array of dimension (*lld_a*, *LOCc(ja* +*n*-1)). On entry, this array contains the local pieces of the symmetric distributed matrix sub(*A*).

If *uplo* = 'U', the leading *n*-by-*n* upper triangular part of sub(*A*) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If *uplo* = 'L', the leading *n*-by-*n* lower triangular part of sub(*A*) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. See *Application Notes* below.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local)

REAL for pssytrd

DOUBLE PRECISION for pdsytrd.

Workspace array of dimension *lwork*.

lwork (local or global) INTEGER, dimension of *work*, must be at least:

lwork $\geq \max(\text{NB} * (np + 1), 3 * \text{NB}),$

where NB = *mb_a* = *nb_a*,

np = numroc(*n*, NB, MYROW, *iarow*, NPROW),

iarow = indxg2p(*ia*, NB, MYROW, *rsrc_a*, NPROW).

indxg2p and *numroc* are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

a

On exit, if *uplo* = 'U', the diagonal and first superdiagonal of sub(*A*) are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the orthogonal matrix *Q* as a product of elementary reflectors; if *uplo* = 'L', the diagonal and first subdiagonal of sub(*A*) are overwritten by the

corresponding elements of the tridiagonal matrix T , and the elements below the first subdiagonal, with the array τ , represent the orthogonal matrix Q as a product of elementary reflectors. See *Application Notes* below.

d

(local)

REAL for pssytrd

DOUBLE PRECISION for pdsytrd.

Arrays, size $LOCc(ja+n-1)$. The diagonal elements of the tridiagonal matrix T :

$$d(i) = A(i,i).$$

d is tied to the distributed matrix *A*.

e

(local)

REAL for pssytrd

DOUBLE PRECISION for pdsytrd.

Arrays, size $LOCc(ja+n-1)$ if $uplo = 'U'$, $LOCc(ja+n-2)$ otherwise.

The off-diagonal elements of the tridiagonal matrix T :

$$e(i) = A(i,i+1) \text{ if } uplo = 'U',$$

$$e(i) = A(i+1,i) \text{ if } uplo = 'L'.$$

e is tied to the distributed matrix *A*.

tau

(local)

REAL for pssytrd

DOUBLE PRECISION for pdsytrd.

Arrays, size $LOCc(ja+n-1)$. This array contains the scalar factors *tau* of the elementary reflectors. *tau* is tied to the distributed matrix *A*.

work(1)

On exit *work(1)* contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i** 100 + *j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) \dots H(2) H(1).$$

Each $H(i)$ has the form

$$H(i) = i - \tau * v * v',$$

where τ is a real scalar, and v is a real vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and τ in $\tau(ja+i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$Q = H(1) \ H(2) \dots \ H(n-1)$.

Each $H(i)$ has the form

$H(i) = i - tau * v * v'$,

where tau is a real scalar, and v is a real vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and tau in $\tau_{\text{au}}(ja+i-1)$.

The contents of $\text{sub}(A)$ on exit are illustrated by the following examples with $n = 5$:

If $\text{uplo} = 'U'$:

$$\begin{array}{cccccc} \dots & \dots & \dots & \dots & \dots & \dots \\ | & & & & & | \\ \dots & \dots & \dots & \dots & \dots & \dots \end{array}$$

If $\text{uplo} = 'L'$:

$$\begin{array}{cccccc} \dots & \dots & \dots & \dots & \dots & \dots \\ | & & & & & | \\ \dots & \dots & \dots & \dots & \dots & \dots \end{array}$$

where d and e denote diagonal and off-diagonal elements of T , and v_i denotes an element of the vector defining $H(i)$.

p?ormtr

Multiples a general matrix by the orthogonal transformation matrix from a reduction to tridiagonal form determined by p?syrtd.

Syntax

Fortran:

```
call psormtr(side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descC, work, lwork, info)
```

```
call pdormtr(side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descC, work, lwork, info)
```

C:

```
void psormtr (char *side , char *uplo , char *trans , MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descC , float *work , MKL_INT *lwork , MKL_INT *info );
void pdormtr (char *side , char *uplo , char *trans , MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descC , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This routine overwrites the general real distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q^* \text{sub}(C)$
<i>trans</i> = 'T':	$Q^T * \text{sub}(C)$

where Q is a real orthogonal distributed matrix of order nq , with $nq = m$ if *side* = 'L' and $nq = n$ if *side* = 'R'.

Q is defined as the product of nq elementary reflectors, as returned by p?sytrd.

If *uplo* = 'U', $Q = H(nq-1) \dots H(2) H(1)$;

If *uplo* = 'L', $Q = H(1) H(2) \dots H(nq-1)$.

Input Parameters

<i>side</i>	(global) CHARACTER
= 'L':	Q or Q^T is applied from the left.
= 'R':	Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER
= 'N',	no transpose, Q is applied.
= 'T',	transpose, Q^T is applied.
<i>uplo</i>	(global) CHARACTER.
= 'U':	Upper triangle of $A(ia:*, ja:*)$ contains elementary reflectors from p?sytrd;
= 'L':	Lower triangle of $A(ia:*, ja:*)$ contains elementary reflectors from p?sytrd
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<i>a</i>	(local) REAL for psormtr DOUBLE PRECISION for pdormtr. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+m-1)</i>) if <i>side</i> ='L', or (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>) if <i>side</i> = 'R'. Contains the vectors which define the elementary reflectors, as returned by p?sytrd. If <i>side</i> ='L', <i>lld_a</i> $\geq \max(1, LOCr(ia+m-1))$;

If $\text{side} = 'R'$, $\text{lld}_a \geq \max(1, \text{LOCr}(ia+n-1))$.

ia, ja
(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau
(local)
REAL for psormtr
DOUBLE PRECISION for pdormtr.
Array, size of *ltau* where
if $\text{side} = 'L'$ and $\text{uplo} = 'U'$, $\text{ltau} = \text{LOCc}(m_a)$,
if $\text{side} = 'L'$ and $\text{uplo} = 'L'$, $\text{ltau} = \text{LOCc}(ja+m-2)$,
if $\text{side} = 'R'$ and $\text{uplo} = 'U'$, $\text{ltau} = \text{LOCc}(n_a)$,
if $\text{side} = 'R'$ and $\text{uplo} = 'L'$, $\text{ltau} = \text{LOCc}(ja+n-2)$.
tau(i) must contain the scalar factor of the elementary reflector *H(i)*, as returned by p?sytrd. *tau* is tied to the distributed matrix *A*.

c
(local) REAL for psormtr
DOUBLE PRECISION for pdormtr.
Pointer into the local memory to an array of dimension (*lld_a*, *LOCc(ja+n-1)*). Contains the local pieces of the distributed matrix sub (*C*).

ic, jc
(global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descC
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work
(local)
REAL for psormtr
DOUBLE PRECISION for pdormtr.
Workspace array of dimension *lwork*.

lwork
(local or global) INTEGER, dimension of *work*, must be at least:
if $\text{uplo} = 'U'$,
 $\text{iaa} = ia; \text{jaa} = ja+1, \text{icc} = ic; \text{jcc} = jc;$
else $\text{uplo} = 'L'$,
 $\text{iaa} = ia+1, \text{jaa} = ja;$
If $\text{side} = 'L'$,
 $\text{icc} = ic+1; \text{jcc} = jc;$
else $\text{icc} = ic; \text{jcc} = jc+1;$

```

    end if
    end if
    If side = 'L',
      mi= m-1; ni= n
      lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 + mpc0)*nb_a) +
      nb_a*nb_a
    else
      If side = 'R',
        mi= m; mi = n-1;
        lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 +
        max(npa0+numroc(numroc(ni+icooffc, nb_a, 0, 0, NPCOL), nb_a,
        0, 0, lcmq), mpc0))*nb_a)+ nb_a*nb_a
      end if
      where lcmq = lcm/NPCOL with lcm = ilcm(NPROW, NPCOL),
            iroffa = mod(iaa-1, mb_a),
            icoffa = mod(jaa-1, nb_a),
            iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
            npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
            iroffc = mod(jcc-1, mb_c),
            icooffc = mod(jcc-1, nb_c),
            icrow = indxg2p(jcc, mb_c, MYROW, rsrc_c, NPROW),
            iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
            mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
            nqc0 = numroc(ni+icooffc, nb_c, MYCOL, iccol, NPCOL),
            ilcm, indxg2p and numroc are ScalAPACK tool functions; MYROW, MYCOL,
            NPROW and NPCOL can be determined by calling the subroutine
            blacs_gridinfo. If lwork = -1, then lwork is global input and a
            workspace query is assumed; the routine only calculates the minimum and
            optimal size for all work arrays. Each of these values is returned in the first
            entry of the corresponding work array, and no error message is issued by
            pxerbla.

```

Output Parameters

<i>c</i>	Overwritten by the product $Q * \text{sub}(C)$, or $Q' * \text{sub}(C)$, or $\text{sub}(C) * Q'$, or $\text{sub}(C) * Q$.
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful.

< 0 : if the i -th argument is an array and the j -entry had an illegal value, then $\text{info} = - (i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.

p?hetrd

Reduces a Hermitian matrix to Hermitian tridiagonal form by a unitary similarity transformation.

Syntax

Fortran:

```
call pchetrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pzhetrd(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

C:

```
void pchetrd (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *d , float *e , MKL_Complex8 *tau , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );
void pzhetrd (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *d , double *e , MKL_Complex16 *tau , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?hetrd routine reduces a complex Hermitian matrix $\text{sub}(A)$ to Hermitian tridiagonal form T by a unitary similarity transformation:

$$Q' \text{sub}(A) Q = T$$

where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

<i>uplo</i>	(global) CHARACTER.
	Specifies whether the upper or lower triangular part of the Hermitian matrix $\text{sub}(A)$ is stored:
	If <i>uplo</i> = 'U', upper triangular
	If <i>uplo</i> = 'L', lower triangular
<i>n</i>	(global) INTEGER. The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
<i>a</i>	(local) COMPLEX for pchetrd DOUBLE COMPLEX for pzhetrd. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). On entry, this array contains the local pieces of the Hermitian distributed matrix $\text{sub}(A)$.

If $uplo = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. (see *Application Notes* below).

ia, ja

(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

work

(local)

COMPLEX for *pchetr**d*

DOUBLE COMPLEX for *pzhetr**d*.

Workspace array of dimension *lwork*.

lwork

(local or global) INTEGER, dimension of *work*, must be at least:

lwork $\geq \max(NB * (np + 1), 3 * NB)$

where $NB = mb_a = nb_a$,

$np = \text{numroc}(n, NB, MYROW, iarow, NPROW)$,

$iarow = \text{indxg2p}(ia, NB, MYROW, rsrc_a, NPROW)$.

indxg2p and *numroc* are ScalAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pixerbla*.

Output Parameters

a

On exit,

If $uplo = 'U'$, the diagonal and first superdiagonal of $\text{sub}(A)$ are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements above the first superdiagonal, with the array *tau*, represent the unitary matrix *Q* as a product of elementary reflectors; if $uplo = 'L'$, the diagonal and first subdiagonal of $\text{sub}(A)$ are overwritten by the corresponding elements of the tridiagonal matrix *T*, and the elements below the first subdiagonal, with the array *tau*, represent the unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

d

(local)

REAL for *pchetr**d*

DOUBLE PRECISION for *pzhetr**d*.

Arrays, size $LOCc(ja+n-1)$. The diagonal elements of the tridiagonal matrix *T*:

$d(i) = A(i,i).$
 d is tied to the distributed matrix A .

e (local)
REAL for pchetrd
DOUBLE PRECISION for pzhetrd.
Arrays, size $LOCc(ja+n-1)$ if $uplo = 'U'$; $LOCc(ja+n-2)$ - otherwise.
The off-diagonal elements of the tridiagonal matrix T :
 $e(i) = A(i,i+1)$ if $uplo = 'U'$,
 $e(i) = A(i+1,i)$ if $uplo = 'L'$.
 e is tied to the distributed matrix A .

tau (local) COMPLEX for pchetrd
DOUBLE COMPLEX for pzhetrd.
Arrays, size $LOCc(ja+n-1)$. This array contains the scalar factors tau of the elementary reflectors. tau is tied to the distributed matrix A .

$work(1)$ On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.

$info$ (global) INTEGER.
= 0: the execution is successful.
< 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = - (i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

If $uplo = 'U'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) * \dots * H(2) * H(1).$$

Each $H(i)$ has the form

$$H(i) = i - tau * v * v^T,$$

where tau is a complex scalar, and v is a complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and tau in $tau(ja+i-1)$.

If $uplo = 'L'$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(n-1).$$

Each $H(i)$ has the form

$$H(i) = i - tau * v * v^T,$$

where tau is a complex scalar, and v is a complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and tau in $tau(ja+i-1)$.

The contents of $\text{sub}(A)$ on exit are illustrated by the following examples with $n = 5$:

If $uplo = 'U'$:

$$\begin{bmatrix} d & e & v_2 & v_3 & v_4 \\ & d & e & v_3 & v_4 \\ & & d & e & v_4 \\ & & & d & e \\ & & & & d \end{bmatrix}$$

If `uplo = 'L'`:

$$\begin{bmatrix} d & & & & \\ e & d & & & \\ v_1 & e & d & & \\ v_1 & v_2 & e & d & \\ v_1 & v_2 & v_3 & e & d \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of T , and v_i denotes an element of the vector defining $H(i)$.

p?unmtr

Multiples a general matrix by the unitary transformation matrix from a reduction to tridiagonal form determined by p?hetrd.

Syntax

Fortran:

```
call pcunmtr(side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
call pzunmtr(side, uplo, trans, m, n, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

C:

```
void pcunmtr (char *side , char *uplo , char *trans , MKL_INT *m , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );
void pzunmtr (char *side , char *uplo , char *trans , MKL_INT *m , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This routine overwrites the general complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

`side = 'L'`

`side = 'R'`

<i>trans</i> = 'N':	$Q^* \text{sub}(C)$	$\text{sub}(C)^* Q$
<i>trans</i> = 'C':	$Q^H \text{sub}(C)$	$\text{sub}(C)^* Q^H$

where Q is a complex unitary distributed matrix of order nq , with $nq = m$ if $\text{side} = 'L'$ and $nq = n$ if $\text{side} = 'R'$.

Q is defined as the product of $nq-1$ elementary reflectors, as returned by [p?hetrd](#).

If $\text{uplo} = 'U'$, $Q = H(nq-1) \dots H(2) H(1)$;

If $\text{uplo} = 'L'$, $Q = H(1) H(2) \dots H(nq-1)$.

Input Parameters

<i>side</i>	(global) CHARACTER = 'L': Q or Q^H is applied from the left. = 'R': Q or Q^H is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'C', conjugate transpose, Q^H is applied.
<i>uplo</i>	(global) CHARACTER. = 'U': Upper triangle of $A(ia:*, ja:*)$ contains elementary reflectors from p?hetrd ; = 'L': Lower triangle of $A(ia:*, ja:*)$ contains elementary reflectors from p?hetrd
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
<i>a</i>	(local) REAL for pcunmtr DOUBLE PRECISION for pzunmtr . Pointer into the local memory to an array of dimension $(\text{lld}_a, \text{LOCc}(ja+m-1))$ if $\text{side} = 'L'$, or $(\text{lld}_a, \text{LOCc}(ja+n-1))$ if $\text{side} = 'R'$. Contains the vectors which define the elementary reflectors, as returned by p?hetrd . If $\text{side} = 'L'$, $\text{lld}_a \geq \max(1, \text{LOCr}(ia+m-1))$; If $\text{side} = 'R'$, $\text{lld}_a \geq \max(1, \text{LOCr}(ia+n-1))$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension $(dlen_)$. The array descriptor for the distributed matrix <i>A</i> .

tau

(local)

COMPLEX for pcunmtr

DOUBLE COMPLEX for pzunmtr.

Array, size of *ltau* where

```
If side = 'L' and uplo = 'U', ltau = LOCC(m_a),
if side = 'L' and uplo = 'L', ltau = LOCC(ja+m-2),
if side = 'R' and uplo = 'U', ltau = LOCC(n_a),
if side = 'R' and uplo = 'L', ltau = LOCC(ja+n-2).
```

tau(*i*) must contain the scalar factor of the elementary reflector $H(i)$, as returned by p?hetrd. *tau* is tied to the distributed matrix *A*.

c

(local) COMPLEX for pcunmtr

DOUBLE COMPLEX for pzunmtr.

Pointer into the local memory to an array of dimension (*lld_a*, LOCC(ja +
n-1)). Contains the local pieces of the distributed matrix sub (*C*).

ic, jc

(global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descC

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work

(local)

COMPLEX for pcunmtr

DOUBLE COMPLEX for pzunmtr.

Workspace array of dimension *lwork*.*lwork*(local or global) INTEGER, dimension of *work*, must be at least:

```
If uplo = 'U',
iaa= ia; jaa= ja+1, icc= ic; jcc= jc;
else uplo = 'L',
iaa= ia+1, jaa= ja;
If side = 'L',
icc= ic+1; jcc= jc;
else icc= ic; jcc= jc+1;
end if
end if
If side = 'L',
mi= m-1; ni= n
lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 + mpc0)*nb_a) +
nb_a*nb_a
```

```

else
  If side = 'R',
    mi= m; mi = n-1;
    lwork  $\geq \max((nb\_a * (nb\_a - 1)) / 2, (nqc0 +$ 
     $\max(npa0 + \text{numroc}(\text{numroc}(ni + icoffc, nb\_a, 0, 0, \text{NPCOL}), nb\_a,$ 
     $0, 0, lcmq), mpc0) * nb\_a) + nb\_a * nb\_a$ 
  end if
  where lcmq = lcm/NPCOL with lcm = ilcm(NPROW, NPCOL),
  iroffa = mod(iaa-1, mb_a),
  icooffa = mod(jaa-1, nb_a),
  iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
  npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
  iroffc = mod(icc-1, mb_c),
  icooffc = mod(jcc-1, nb_c),
  icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
  iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
  mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
  nqc0 = numroc(ni+icooffc, nb_c, MYCOL, iccol, NPCOL),
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL,
NPROW and NPCOL can be determined by calling the subroutine
blacs_gridinfo. If lwork = -1, then lwork is global input and a
workspace query is assumed; the routine only calculates the minimum and
optimal size for all work arrays. Each of these values is returned in the first
entry of the corresponding work array, and no error message is issued by
pxerbla.

```

Output Parameters

c

Overwritten by the product $Q^* \text{sub}(C)$, or $Q'^* \text{sub}(C)$, or $\text{sub}(C)^* Q'$, or $\text{sub}(C)^* Q$.

work(1)

On exit *work(1)* contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - *i*

p?stebz

Computes the eigenvalues of a symmetric tridiagonal matrix by bisection.

Syntax

Fortran:

```
call psstebz(ictxt, range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w,
iblock, isplit, work, iwork, liwork, info)
```

```
call pdstebz(ictxt, range, order, n, vl, vu, il, iu, abstol, d, e, m, nsplit, w,
iblock, isplit, work, iwork, liwork, info)
```

C:

```
void psstebz (MKL_INT *ictxt , char *range , char *order , MKL_INT *n , float *vl ,
float *vu , MKL_INT *il , MKL_INT *iu , float *abstol , float *d , float *e , MKL_INT
*m , MKL_INT *nsplit , float *w , MKL_INT *iblock , MKL_INT *isplit , float *work ,
MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
```

```
void pdstebz (MKL_INT *ictxt , char *range , char *order , MKL_INT *n , double *vl ,
double *vu , MKL_INT *il , MKL_INT *iu , double *abstol , double *d , double *e ,
MKL_INT *m , MKL_INT *nsplit , double *w , MKL_INT *iblock , MKL_INT *isplit , double
*work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?stebz routine computes the eigenvalues of a symmetric tridiagonal matrix in parallel. These may be all eigenvalues, all eigenvalues in the interval [vl vu], or the eigenvalues indexed *il* through *iu*. A static partitioning of work is done at the beginning of p?stebz which results in all processes finding an (almost) equal number of eigenvalues.

Input Parameters

<i>ictxt</i>	(global) INTEGER. The BLACS context handle.
<i>range</i>	(global) CHARACTER. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues in the interval [vl, vu]. If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> to <i>iu</i> .
<i>order</i>	(global) CHARACTER. Must be 'B' or 'E'. If <i>order</i> = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block. If <i>order</i> = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.
<i>n</i>	(global) INTEGER. The order of the tridiagonal matrix <i>T</i> (<i>n</i> ≥0).
<i>vl</i> , <i>vu</i>	(global) REAL for psstebz DOUBLE PRECISION for pdstebz. If <i>range</i> = 'V', the routine computes the lower and the upper bounds for the eigenvalues on the interval [1, vu]. If <i>range</i> = 'A' or 'I', <i>vl</i> and <i>vu</i> are not referenced.

il, iu

(global)

INTEGER. Constraint: $1 \leq il \leq iu \leq n$.

If $range = 'I'$, the index of the smallest eigenvalue is returned for il and of the largest eigenvalue for iu (assuming that the eigenvalues are in ascending order) must be returned. il must be at least 1. iu must be at least il and no greater than n .

If $range = 'A'$ or ' V ', il and iu are not referenced.

abstol

(global)

REAL for `psstebz`DOUBLE PRECISION for `pdstebz`.

The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width $abstol$. If $abstol \leq 0$, then the tolerance is taken as $ulp \cdot \|T\|_1$, where ulp is the machine precision, and $\|T\|_1$ means the 1-norm of T .

Eigenvalues will be computed most accurately when $abstol$ is set to the underflow threshold `slamch('U')`, not 0. Note that if eigenvectors are desired later by inverse iteration (`p?stein`), $abstol$ should be set to $2 * p?lamch('S')$.

d

(global)

REAL for `psstebz`DOUBLE PRECISION for `pdstebz`.Array, size (n).

Contains n diagonal elements of the tridiagonal matrix T . To avoid overflow, the matrix must be scaled so that its largest entry is no greater than the $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

e

(global)

REAL for `psstebz`DOUBLE PRECISION for `pdstebz`.Array, size ($n - 1$).

Contains $(n-1)$ off-diagonal elements of the tridiagonal matrix T . To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

work

(local)

REAL for `psstebz`DOUBLE PRECISION for `pdstebz`.Array, size $\max(5n, 7)$. This is a workspace array.*lwork*(local) INTEGER. The size of the *work* array must be $\geq \max(5n, 7)$.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

iwork(local) INTEGER. Array, size $\max(4n, 14)$. This is a workspace array.*liwork*(local) INTEGER. the size of the *iwork* array must $\geq \max(4n, 14, \text{NPROCS})$.

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

m(global) INTEGER. The actual number of eigenvalues found. $0 \leq m \leq n$ *nsplit*(global) INTEGER. The number of diagonal blocks detected in T .
 $1 \leq nsplit \leq n$ *w*

(global)

REAL for `psstebz`DOUBLE PRECISION for `pdstebz`.Array, size (n). On exit, the first m elements of *w* contain the eigenvalues on all processes.*iblock*

(global) INTEGER.

Array, size (n). At each row/column j where $e(j)$ is zero or small, the matrix T is considered to split into a block diagonal matrix. On exit $iblock(i)$ specifies which block (from 1 to the number of blocks) the eigenvalue $w(i)$ belongs to.

NOTE

In the (theoretically impossible) event that bisection does not converge for some or all eigenvalues, *info* is set to 1 and the ones for which it did not are identified by a negative block number.

isplit

(global) INTEGER.

Array, size (n).Contains the splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to $isplit(1)$, the second of rows/columns $isplit(1)+1$ through $isplit(2)$, etc., and the $nsplit$ -th consists of rows/columns $isplit(nsplit-1)+1$ through $isplit(nsplit)=n$. (Only the first $nsplit$ elements are used, but since the $nsplit$ values are not known, n words must be reserved for *isplit*.)*info*

(global) INTEGER.

If $info = 0$, the execution is successful.

If $\text{info} < 0$, if $\text{info} = -i$, the i -th argument has an illegal value.

If $\text{info} > 0$, some or all of the eigenvalues fail to converge or not computed.

If $\text{info} = 1$, bisection fails to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.

If $\text{info} = 2$, mismatch between the number of eigenvalues output and the number desired.

If $\text{info} = 3$: $\text{range} = 'i'$, and the Gershgorin interval initially used is incorrect. No eigenvalues are computed. Probable cause: the machine has a sloppy floating point arithmetic. Increase the *fudge* parameter, recompile, and try again.

p?stein

Computes the eigenvectors of a tridiagonal matrix using inverse iteration.

Syntax

Fortran:

```
call psstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork,
iwork, liwork, ifail, iclustr, gap, info)

call pdstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork,
iwork, liwork, ifail, iclustr, gap, info)

call pcstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork,
iwork, liwork, ifail, iclustr, gap, info)

call pzstein(n, d, e, m, w, iblock, isplit, orfac, z, iz, jz, descz, work, lwork,
iwork, liwork, ifail, iclustr, gap, info)
```

C:

```
void psstein (MKL_INT *n , float *d , float *e , MKL_INT *m , float *w , MKL_INT
*iblock , MKL_INT *isplit , float *orfac , float *z , MKL_INT *iz , MKL_INT *jz ,
MKL_INT *descz , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT *info );

void pdstein (MKL_INT *n , double *d , double *e , MKL_INT *m , double *w , MKL_INT
*iblock , MKL_INT *isplit , double *orfac , double *z , MKL_INT *iz , MKL_INT *jz ,
MKL_INT *descz , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *ifail , MKL_INT *iclustr , double *gap , MKL_INT *info );

void pcstein (MKL_INT *n , float *d , float *e , MKL_INT *m , float *w , MKL_INT
*iblock , MKL_INT *isplit , float *orfac , MKL_Complex8 *z , MKL_INT *iz , MKL_INT
*jz , MKL_INT *descz , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT
*liwork , MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT *info );

void pzstein (MKL_INT *n , double *d , double *e , MKL_INT *m , double *w , MKL_INT
*iblock , MKL_INT *isplit , double *orfac , MKL_Complex16 *z , MKL_INT *iz , MKL_INT
*jz , MKL_INT *descz , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT
*liwork , MKL_INT *ifail , MKL_INT *iclustr , double *gap , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?stein` routine computes the eigenvectors of a symmetric tridiagonal matrix T corresponding to specified eigenvalues, by inverse iteration. `p?stein` does not orthogonalize vectors that are on different processes. The extent of orthogonalization is controlled by the input parameter `lwork`. Eigenvectors that are to be orthogonalized are computed by the same process. `p?stein` decides on the allocation of work among the processes and then calls `?stein2` (modified LAPACK routine) on each individual process. If insufficient workspace is allocated, the expected orthogonalization may not be done.

NOTE

If the eigenvectors obtained are not orthogonal, increase `lwork` and run the code again.

`p = NPROW*NPCOL` is the total number of processes.

Input Parameters

<code>n</code>	(global) INTEGER. The order of the matrix T ($n \geq 0$).
<code>m</code>	(global) INTEGER. The number of eigenvectors to be returned.
<code>d, e, w</code>	<p>(global)</p> <p>REAL for single-precision flavors</p> <p>DOUBLE PRECISION for double-precision flavors.</p> <p>Arrays: <code>d(*)</code> contains the diagonal elements of T.</p> <p><code>size (n)</code>.</p> <p><code>e(*)</code> contains the off-diagonal elements of T.</p> <p><code>size (n-1)</code>.</p> <p><code>w(*)</code> contains all the eigenvalues grouped by split-off block. The eigenvalues are supplied from smallest to largest within the block. (Here the output array <code>w</code> from <code>p?stebz</code> with <code>order = 'B'</code> is expected. The array should be replicated in all processes.)</p> <p><code>size(m)</code></p>
<code>iblock</code>	<p>(global) INTEGER.</p> <p>Array, <code>size (n)</code>. The submatrix indices associated with the corresponding eigenvalues in <code>w--1</code> for eigenvalues belonging to the first submatrix from the top, 2 for those belonging to the second submatrix, etc. (The output array <code>iblock</code> from <code>p?stebz</code> is expected here).</p>
<code>isplit</code>	<p>(global) INTEGER.</p> <p>Array, <code>size (n)</code>. The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to <code>isplit(1)</code>, the second of rows/columns <code>isplit(1)+1</code> through <code>isplit(2)</code>, etc., and the <code>nsplit</code>-th consists of rows/columns <code>isplit(nsplits-1)+1</code> through <code>isplit(nsplits)=n</code>. (The output array <code>isplit</code> from <code>p?stebz</code> is expected here.)</p>

<i>orfac</i>	(global) REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors.
	<i>orfac</i> specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues within $\text{orfac} * \mathbf{T} $ of each other are to be orthogonalized. However, if the workspace is insufficient (see <i>lwork</i>), this tolerance may be decreased until all eigenvectors can be stored in one process. No orthogonalization is done if <i>orfac</i> is equal to zero. A default value of 1000 is used if <i>orfac</i> is negative. <i>orfac</i> should be identical on all processes
<i>iz, jz</i>	(global) INTEGER. The row and column indices in the global array <i>z</i> indicating the first row and the first column of the submatrix <i>Z</i> , respectively.
<i>descz</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>Z</i> .
<i>work</i>	(local). REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Workspace array, size (<i>lwork</i>).
<i>lwork</i>	(local) INTEGER. <i>lwork</i> controls the extent of orthogonalization which can be done. The number of eigenvectors for which storage is allocated on each process is $nvec = \text{floor}((lwork - \max(5*n, np00*mq00))/n)$. Eigenvectors corresponding to eigenvalue clusters of size <i>nvec</i> - $\text{ceil}(m/p)$ + 1 are guaranteed to be orthogonal (the orthogonality is similar to that obtained from ?stein2).

NOTE

lwork must be no smaller than $\max(5*n, np00*mq00) + \text{ceil}(m/p)*n$ and should have the same input value on all processes.

	It is the minimum value of <i>lwork</i> input on different processes that is significant.
	If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.
<i>iwork</i>	(local) INTEGER. Workspace array, size $(3n+p+1)$.
<i>liwork</i>	(local) INTEGER. The size of the array <i>iwork</i> . It must be greater than $(3*n + p + 1)$.

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

`z`

(local)
REAL for `psstein`
DOUBLE PRECISION for `pdstein`
COMPLEX for `pcstein`
DOUBLE COMPLEX for `pzstein`.

Array, size (`descz(dlen_)`, $n/\text{NPCOL} + \text{NB}$). `z` contains the computed eigenvectors associated with the specified eigenvalues. Any vector which fails to converge is set to its current iterate after MAXIT iterations (See [?stein2](#)). On output, `z` is distributed across the p processes in block cyclic format.

`work(1)`

On exit, `work(1)` gives a lower bound on the workspace ($lwork$) that guarantees the user desired orthogonalization (see `orfac`). Note that this may overestimate the minimum workspace needed.

`iwork`

On exit, `iwork(1)` contains the amount of integer workspace required. On exit, the `iwork(2)` through `iwork(p+2)` indicate the eigenvectors computed by each process. Process i computes eigenvectors indexed $iwork(i+2)+1$ through `iwork(i+3)`.

`ifail`

(global) **INTEGER**. Array, size (m). On normal exit, all elements of `ifail` are zero. If one or more eigenvectors fail to converge after MAXIT iterations (as in [?stein](#)), then `info > 0` is returned. If $\text{mod}(info, m+1) > 0$, then for $i=1$ to $\text{mod}(info, m+1)$, the eigenvector corresponding to the eigenvalue $w(ifail(i))$ failed to converge (w refers to the array of eigenvalues on output).

`iclustr`

(global) **INTEGER**. Array, size ($2*p$)

This output array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be orthogonalized due to insufficient workspace (see `lwork`, `orfac` and `info`). Eigenvectors corresponding to clusters of eigenvalues indexed `iclustr(2*I-1)` to `iclustr(2*I)`, $i = 1$ to $\text{info}/(m+1)$, could not be orthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. `iclustr` is a zero terminated array ---(`iclustr(2*k).ne. 0.and.iclustr(2*k+1).eq.0`) if and only if k is the number of clusters.

`gap`

(global)

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors.

This output array contains the gap between eigenvalues whose eigenvectors could not be orthogonalized. The info/m output values in this array correspond to the $\text{info}/(m+1)$ clusters indicated by the array iclustr . As a result, the dot product between eigenvectors corresponding to the i -th cluster may be as high as $(O(n) * \text{macheps}) / \text{gap}(i)$.

info

(global) INTEGER.

If $\text{info} = 0$, the execution is successful.If $\text{info} < 0$: If the i -th argument is an array and the j -entry had an illegal value, then $\text{info} = -(i*100+j)$,If the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.If $\text{info} < 0$: if $\text{info} = -i$, the i -th argument had an illegal value.If $\text{info} > 0$: if $\text{mod}(\text{info}, m+1) = i$, then i eigenvectors failed to converge in MAXIT iterations. Their indices are stored in the array ifail . If $\text{info}/(m+1) = i$, then eigenvectors corresponding to i clusters of eigenvalues could not be orthogonalized due to insufficient workspace. The indices of the clusters are stored in the array iclustr .

Nonsymmetric Eigenvalue Problems

This section describes ScaLAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

To solve a nonsymmetric eigenvalue problem with ScaLAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained.

[Table "Computational Routines for Solving Nonsymmetric Eigenproblems"](#) lists ScaLAPACK routines for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A = QHQ^H$, as well as routines for solving eigenproblems with Hessenberg matrices, and multiplying the matrix after reduction.

Computational Routines for Solving Nonsymmetric Eigenproblems

Operation performed	General matrix	Orthogonal/Unitary matrix	Hessenberg matrix
Reduce to Hessenberg form $A = QHQ^H$	p?gehrd		
Multiply the matrix after reduction		p?ormhr / p?unmhr	
Find eigenvalues and Schur factorization			p?lahqr , p?hseqr

[p?gehrd](#)

Reduces a general matrix to upper Hessenberg form.

Syntax

Fortran:

```
call psgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pdgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pcgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pzgehrd(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```

void psgehrd (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT
*info );

void pdgehrd (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT
*info );

void pcgehrd (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzgehrd (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The p?gehrd routine reduces a real/complex general distributed matrix sub (A) to upper Hessenberg form H by an orthogonal or unitary similarity transformation

$$Q^T \text{sub}(A) Q = H,$$

where $\text{sub}(A) = A(i_a+n-1:i_a+n-1, j_a+n-1:j_a+n-1)$.

Input Parameters

n (global) INTEGER. The order of the distributed matrix sub(A) ($n \geq 0$).

ilo, *ihi* (global) INTEGER.

It is assumed that sub(A) is already upper triangular in rows $i_a:i_a+ilo-2$ and $i_a+ihi:i_a+n-1$ and columns $j_a:j_a+ilo-2$ and $j_a+ihi:j_a+n-1$. (See Application Notes below).

If $n > 0$, $1 \leq ilo \leq ihi \leq n$; otherwise set $ilo = 1$, $ihi = n$.

a (local) REAL for psgehrd

DOUBLE PRECISION for pdgehrd

COMPLEX for pcgehrd

DOUBLE COMPLEX for pzgehrd.

Pointer into the local memory to an array of dimension (*lld_a*, *LOCc(ja+n-1)*). On entry, this array contains the local pieces of the n -by- n general distributed matrix sub(A) to be reduced.

ia, *ja*

(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A , respectively.

desca

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A .

work

(local)

REAL for psgehrd
 DOUBLE PRECISION for pdgehrd
 COMPLEX for pcgehrd
 DOUBLE COMPLEX for pzgehrd.
 Workspace array of dimension *lwork*.

lwork

(local or global) INTEGER, dimension of the array *work*. *lwork* is local input and must be at least

```
lwork ≥ NB * NB + NB * max(ihip+1, ihlp+inlq)  

where NB = mb_a = nb_a,  

iroffa = mod(ia-1, NB),  

icoffa = mod(ja-1, NB),  

ioff = mod(ia+ilo-2, NB), iarow = indxg2p(ia, NB, MYROW,  

rsrc_a, NPROW), ihip = numroc(ihi+iroffa, NB, MYROW, iarow,  

NPROW),  

ilrow = indxg2p(ia+ilo-1, NB, MYROW, rsrc_a, NPROW),  

ihlp = numroc(ihi-ilo+ioff+1, NB, MYROW, ilrow, NPROW),  

ilcol = indxg2p(ja+ilo-1, NB, MYCOL, csrc_a, NPCOL),  

inlq = numroc(n-ilo+ioff+1, NB, MYCOL, ilcol, NPCOL),  

indxg2p and numroc are ScalAPACK tool functions; MYROW, MYCOL, NPROW  

and NPCOL can be determined by calling the subroutine blacs_gridinfo.
```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a

On exit, the upper triangle and the first subdiagonal of sub(*A*) are overwritten with the upper Hessenberg matrix *H*, and the elements below the first subdiagonal, with the array *tau*, represent the orthogonal/unitary matrix *Q* as a product of elementary reflectors (see *Application Notes* below).

tau

(local). REAL for psgehrd
 DOUBLE PRECISION for pdgehrd
 COMPLEX for pcgehrd
 DOUBLE COMPLEX for pzgehrd.
 Array, size at least max(*ja*+*n*-2).

The scalar factors of the elementary reflectors (see *Application Notes* below). Elements *ja*:*ja*+*ilo*-2 and *ja*+*ihi*:*ja*+*n*-2 of *tau* are set to zero. *tau* is tied to the distributed matrix *A*.

<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> * 100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Application Notes

The matrix Q is represented as a product of $(ihi-ilo)$ elementary reflectors

$$Q = H(ilo)*H(iло+1)*\dots*H(ihi-1).$$

Each $H(i)$ has the form

$$H(i) = i - tau * v * v'$$

where tau is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$, $v(i+1) = 1$ and $v(ihi+1:n) = 0$; $v(i+2:ihi)$ is stored on exit in $a(ia+ilo+i:ia+ihi-1, ja+ilo+i-2)$, and tau in $\tau(a(ja+ilo+i-2))$. The contents of $a(ia:ia+n-1, ja:ja+n-1)$ are illustrated by the following example, with $n = 7$, $ilo = 2$ and $ihi = 6$:

on entry

$$\begin{bmatrix} a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \end{bmatrix}$$

on exit

$$\begin{bmatrix} a & a & a & h & h & h & a \\ a & h & h & h & h & h & a \\ h & h & h & h & h & h & h \\ v2 & h & h & h & h & h & h \\ v2 & v3 & h & h & h & h & h \\ v2 & v3 & v4 & h & h & h & h \\ & & & & & & a \end{bmatrix}$$

where a denotes an element of the original matrix $\text{sub}(A)$, H denotes a modified element of the upper Hessenberg matrix H , and vi denotes an element of the vector defining $H(ja+ilo+i-2)$.

p?ormhr

Multiples a general matrix by the orthogonal transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

Fortran:

```
call psormhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pdormhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

C:

```
void psormhr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *ilo ,
MKL_INT *ihi , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau ,
float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork ,
MKL_INT *info );
```

```
void pdormhr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *ilo ,
MKL_INT *ihi , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau ,
double *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?ormhr` routine overwrites the general real distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q^* \text{sub}(C)$	$\text{sub}(C)*Q$
<i>trans</i> = 'T':	$Q^T * \text{sub}(C)$	$\text{sub}(C)*Q^T$

where Q is a real orthogonal distributed matrix of order nq , with $nq = m$ if $\text{side} = 'L'$ and $nq = n$ if $\text{side} = 'R'$.

Q is defined as the product of $ihi-ilo$ elementary reflectors, as returned by `p?gehrd`.

$Q = H(i_{lo}) H(i_{lo+1}) \dots H(i_{hi-1})$.

Input Parameters

<i>side</i>	(global) CHARACTER = 'L': Q or Q^T is applied from the left. = 'R': Q or Q^T is applied from the right.
<i>trans</i>	(global) CHARACTER = 'N', no transpose, Q is applied. = 'T', transpose, Q^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).

ilo, ihi

(global) INTEGER.

ilo and *ihi* must have the same values as in the previous call of p?gehrd. *Q* is equal to the unit matrix except for the distributed submatrix $Q(ia + ilo : ia + ihi - 1, ia + ilo : ja + ihi - 1)$.

If *side* = 'L', $1 \leq ilo \leq ihi \leq \max(1, m)$;

If *side* = 'R', $1 \leq ilo \leq ihi \leq \max(1, n)$;

ilo and *ihi* are relative indexes.

a

(local)

REAL for psormhr

DOUBLE PRECISION for pdormhr

Pointer into the local memory to an array of dimension (*lld_a*, *LOCc(ja + m - 1)*) if *side* = 'L', and (*lld_a*, *LOCc(ja + n - 1)*) if *side* = 'R'.

Contains the vectors which define the elementary reflectors, as returned by p?gehrd.

ia, ja

(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau

(local)

REAL for psormhr

DOUBLE PRECISION for pdormhr

Array, size *LOCc(ja + m - 2)*, if *side* = 'L', and *LOCc(ja + n - 2)* if *side* = 'R'.

This array contains the scalar factors *tau(j)* of the elementary reflectors *H(j)* as returned by p?gehrd. *tau* is tied to the distributed matrix *A*.

c

(local)

REAL for psormhr

DOUBLE PRECISION for pdormhr

Pointer into the local memory to an array of dimension (*lld_c*, *LOCc(jc + n - 1)*).

Contains the local pieces of the distributed matrix sub(*C*).

ic, jc

(global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.

descC

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.

work

(local)

REAL for psormhr

DOUBLE PRECISION for pdormhr

Workspace array with dimension *lwork*.

lwork (local or global) INTEGER.

The dimension of the array *work*.

lwork must be at least $iaa = ia + ilo; jaa = ja + ilo - 1$;

If *side* = 'L',

$$mi = ihi - ilo; ni = n; icc = ic + ilo; jcc = jc; lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + mpc0) * nb_a) + nb_a * nb_a$$

else if *side* = 'R',

$$mi = m; ni = ihi - ilo; icc = ic; jcc = jc + ilo; lwork \geq \max((nb_a * (nb_a - 1)) / 2, (nqc0 + \max(npao + \text{numroc}(\text{numroc}(ni + ioffc, nb_a, 0, 0, \text{NPCOL}), nb_a, 0, 0, lcmq), mpc0)) * nb_a) + nb_a * nb_a$$

end if

where $lcmq = lcm/\text{NPCOL}$ with $lcm = \text{ilcm}(\text{NPROW}, \text{NPCOL})$,

$$\text{iroffa} = \text{mod}(iaa - 1, mb_a),$$

$$\text{icoffa} = \text{mod}(jaa - 1, nb_a),$$

$$\text{iarow} = \text{idxg2p}(iaa, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}),$$

$$\text{npao} = \text{numroc}(ni + \text{iroffa}, mb_a, \text{MYROW}, \text{iarow}, \text{NPROW}),$$

$$\text{iroffc} = \text{mod}(icc - 1, mb_c), \text{icoffc} = \text{mod}(jcc - 1, nb_c),$$

$$\text{icrow} = \text{idxg2p}(icc, mb_c, \text{MYROW}, rsrc_c, \text{NPROW}),$$

$$\text{iccol} = \text{idxg2p}(jcc, nb_c, \text{MYCOL}, csrc_c, \text{NPCOL}),$$

$$\text{mpc0} = \text{numroc}(mi + \text{iroffc}, mb_c, \text{MYROW}, \text{icrow}, \text{NPROW}),$$

$$\text{nqc0} = \text{numroc}(ni + \text{icoffc}, nb_c, \text{MYCOL}, \text{iccol}, \text{NPCOL}),$$

ilcm, *idxg2p* and *numroc* are ScaLAPACK tool functions; *MYROW*, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pixerbla*.

Output Parameters

c sub(*C*) is overwritten by $Q * \text{sub}(C)$, or $Q' * \text{sub}(C)$, or $\text{sub}(C) * Q'$, or $\text{sub}(C) * Q$.

work(1) On exit *work(1)* contains the minimum value of *lwork* required for optimum performance.

info (global) INTEGER.

= 0: the execution is successful.

< 0 : if the i -th argument is an array and the j -entry had an illegal value, then $\text{info} = - (i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.

p?unmhr

Multiples a general matrix by the unitary transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

Fortran:

```
call pcunmhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
call pzunmhr(side, trans, m, n, ilo, ihi, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

C:

```
void pcunmhr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *ilo ,
MKL_INT *ihi , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_Complex8 *tau , MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );
void pzunmhr (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *ilo ,
MKL_INT *ihi , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_Complex16 *tau , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This routine overwrites the general complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$\text{side} = 'L'$	$\text{side} = 'R'$
$\text{trans} = 'N':$	$Q^* \text{sub}(C)$	$\text{sub}(C)*Q$
$\text{trans} = 'H':$	$Q^H * \text{sub}(C)$	$\text{sub}(C)*Q^H$

where Q is a complex unitary distributed matrix of order nq , with $nq = m$ if $\text{side} = 'L'$ and $nq = n$ if $\text{side} = 'R'$.

Q is defined as the product of $ihi-ilo$ elementary reflectors, as returned by [p?gehrd](#).

$Q = H(i_{lo}) H(i_{lo}+1) \dots H(i_{hi}-1)$.

Input Parameters

side	(global) CHARACTER
$= 'L':$	Q or Q^H is applied from the left.
$= 'R':$	Q or Q^H is applied from the right.
trans	(global) CHARACTER

= 'N', no transpose, Q is applied.

= 'C', conjugate transpose, Q^H is applied.

m
(global) INTEGER. The number of rows in the distributed submatrix sub (C)
($m \geq 0$).

n
(global) INTEGER. The number of columns in the distributed submatrix sub
(C) ($n \geq 0$).

ilo, ihi
(global) INTEGER

These must be the same parameters *ilo* and *ihi*, respectively, as supplied to p?gehrd. Q is equal to the unit matrix except in the distributed submatrix Q ($ia+ilo:ia+ihi-1, ia+ilo:ja+ihi-1$).

If *side* = 'L', then $1 \leq ilo \leq ihi \leq \max(1, m)$.

If *side* = 'R', then $1 \leq ilo \leq ihi \leq \max(1, n)$

ilo and *ihi* are relative indexes.

a
(local)

COMPLEX for pcunmhr

DOUBLE COMPLEX for pzunmhr.

Pointer into the local memory to an array of dimension (*lld_a*, LOC $c(ja+m-1)$) if *side* = 'L', and (*lld_a*, LOC $c(ja+n-1)$) if *side* = 'R'.

Contains the vectors which define the elementary reflectors, as returned by p?gehrd.

ia, ja
(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca
(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

tau
(local)

COMPLEX for pcunmhr

DOUBLE COMPLEX for pzunmhr.

Array, size LOC $c(ja+m-2)$, if *side* = 'L', and LOC $c(ja+n-2)$ if *side* = 'R'.

This array contains the scalar factors $\tau(j)$ of the elementary reflectors $H(j)$ as returned by p?gehrd. *tau* is tied to the distributed matrix *A*.

c
(local)

COMPLEX for pcunmhr

DOUBLE COMPLEX for pzunmhr.

Pointer into the local memory to an array of dimension (*lld_c*, LOC $c(jc+n-1)$).

Contains the local pieces of the distributed matrix sub(*C*).

<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descC</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) COMPLEX for <i>pcunmhr</i> DOUBLE COMPLEX for <i>pzunmhr</i> . Workspace array with dimension <i>lwork</i> .
<i>lwork</i>	(local or global) The dimension of the array <i>work</i> . <i>lwork</i> must be at least <i>iaa</i> = <i>ia</i> + <i>ilo</i> ; <i>jaa</i> = <i>ja</i> + <i>ilo</i> -1; If <i>side</i> = 'L', <i>mi</i> = <i>ihi</i> - <i>ilo</i> ; <i>ni</i> = <i>n</i> ; <i>icc</i> = <i>ic</i> + <i>ilo</i> ; <i>jcc</i> = <i>jc</i> ; <i>lwork</i> $\geq \max((nb_a * (nb_a-1))/2, (nqc0+mpc0)*nb_a) + nb_a * nb_aelse if side = 'R',mi = m; ni = ihi-ilo; icc = ic; jcc = jc + ilo; lwork \geq\max((nb_a * (nb_a-1))/2, (nqc0 + \max(npa0+numroc(numroc(ni+icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0)) *nb_a) + nb_a * nb_aend ifwhere lcmq = lcm/NPCOL with lcm = ilcm(NPROW, NPCOL),iroffa = mod(iaa-1, mb_a),icoffa = mod(jaa-1, nb_a),iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),iroffc = mod(icc-1, mb_c),icoffc = mod(jcc-1, nb_c),icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),ilcm, indxg2p and numroc are ScalAPACK tool functions; MYROW, MYCOL,NPROW and NPCOL can be determined by calling the subroutineblacs_gridinfo.If lwork = -1, then lwork is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.$

Output Parameters

<i>c</i>	<i>C</i> is overwritten by $Q^* \text{sub}(C)$ or $Q'^*\text{sub}(C)$ or $\text{sub}(C)*Q'$ or $\text{sub}(C)*Q$.
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> * 100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?lahqr

Computes the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form.

Syntax

Fortran:

```
call pslahqr(wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, ilwork, info)
call pdlahqr(wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, ilwork, info)
```

C:

```
void pslahqr (MKL_INT *wantt , MKL_INT *wantz , MKL_INT *n , MKL_INT *ilo , MKL_INT
*ihi , float *a , MKL_INT *desca , float *wr , float *wi , MKL_INT *iloz , MKL_INT
*ihiz , float *z , MKL_INT *descz , float *work , MKL_INT *lwork , MKL_INT *iwork ,
MKL_INT *ilwork , MKL_INT *info );
void pdlahqr (MKL_INT *wantt , MKL_INT *wantz , MKL_INT *n , MKL_INT *ilo , MKL_INT
*ihi , double *a , MKL_INT *desca , double *wr , double *wi , MKL_INT *iloz , MKL_INT
*ihiz , double *z , MKL_INT *descz , double *work , MKL_INT *lwork , MKL_INT *iwork ,
MKL_INT *ilwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This is an auxiliary routine used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns *ilo* to *ihi*.

Input Parameters

<i>wantt</i>	(global) LOGICAL If <i>wantt</i> = .TRUE., the full Schur form <i>T</i> is required; If <i>wantt</i> = .FALSE., only eigenvalues are required.
<i>wantz</i>	(global) LOGICAL. If <i>wantz</i> = .TRUE., the matrix of Schur vectors <i>z</i> is required;

If `wantz` = .FALSE., Schur vectors are not required.

`n` (global) INTEGER. The order of the Hessenberg matrix A (and z if `wantz`). ($n \geq 0$).

`ilo, ihi` (global) INTEGER.

It is assumed that A is already upper quasi-triangular in rows and columns $ihi+1:n$, and that $A(ilo, ilo-1) = 0$ (unless $ilo = 1$). `p?lahqr` works primarily with the Hessenberg submatrix in rows and columns ilo to ihi , but applies transformations to all of A if `wantt` is .TRUE.. $1 \leq ilo \leq \max(1, ihi); ihi \leq n$.

`a` (global)

REAL for `pslahqr`

DOUBLE PRECISION for `pdlahqr`

Array, DIMENSION (`descA(1ld_)`, *). On entry, the upper Hessenberg matrix A .

`descA` (global and local) INTEGER array, dimension (`dlen_`). The array descriptor for the distributed matrix A .

`iloz, ihiz` (global) INTEGER. Specify the rows of z to which transformations must be applied if `wantz` is .TRUE.. $1 \leq iloz \leq ilo; ihi \leq ihiz \leq n$.

`z` (global) REAL for `pslahqr`

DOUBLE PRECISION for `pdlahqr`

Array. If `wantz` is .TRUE., on entry z must contain the current matrix Z of transformations accumulated by `pdhseqr`. If `wantz` is .FALSE., z is not referenced.

`descZ` (global and local) INTEGER array, dimension (`dlen_`). The array descriptor for the distributed matrix Z .

`work` (local)

REAL for `pslahqr`

DOUBLE PRECISION for `pdlahqr`

Workspace array with dimension `lwork`.

`lwork` (local) INTEGER. The dimension of `work`. `lwork` is assumed big enough so that $lwork \geq 3*n + \max(2*\max(descZ(1ld_), descA(1ld_)) + 2*LOCq(n), 7*\text{ceil}(n/hbl)/\text{lcm}(NPROW, NPCOL))$.

If `lwork` = -1, then `work(1)` gets set to the above number and the code returns immediately.

`iwork`

(global and local) INTEGER array of size `ilwork`.

`ilwork`

(local) INTEGER This holds some of the $iblk$ integer arrays.

Output Parameters

<i>a</i>	On exit, if <i>wantt</i> is .TRUE., <i>A</i> is upper quasi-triangular in rows and columns <i>ilo:ih</i> , with any 2-by-2 or larger diagonal blocks not yet in standard form. If <i>wantt</i> is .FALSE., the contents of <i>A</i> are unspecified on exit.
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>wr, wi</i>	(global replicated output) REAL for pslahqr DOUBLE PRECISION for pdlahqr Arrays, DIMENSION(<i>n</i>) each. The real and imaginary parts, respectively, of the computed eigenvalues <i>ilo</i> to <i>ih</i> are stored in the corresponding elements of <i>wr</i> and <i>wi</i> . If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of <i>wr</i> and <i>wi</i> , say the <i>i</i> -th and (<i>i</i> +1)-th, with <i>wi(i)>0</i> and <i>wi(i+1)<0</i> . If <i>wantt</i> is .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in <i>A</i> . <i>A</i> may be returned with larger diagonal blocks until the next release.
<i>z</i>	On exit <i>z</i> has been updated; transformations are applied only to the submatrix <i>z(iloz:ihiz, ilo:ih)</i> .
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: parameter number <i>-info</i> incorrect or inconsistent > 0: p?lahqr failed to compute all the eigenvalues <i>ilo</i> to <i>ih</i> in a total of $30 * (ih - ilo + 1)$ iterations; if <i>info = i</i> , elements <i>i+1:ih</i> of <i>wr</i> and <i>wi</i> contain those eigenvalues which have been successfully computed.

p?hseqr

Computes eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form.

Syntax

Fortran:

```
call pshseqr( job, compz, n, ilo, ihi, h, desch, wr, wi, z, descz, work, lwork, iwork, liwork, info )
call pdhseqr( job, compz, n, ilo, ihi, h, desch, wr, wi, z, descz, work, lwork, iwork, liwork, info )
```

Include Files

- C: mkl_scalapack.h

Description

p?hseqr computes the eigenvalues of an upper Hessenberg matrix *H* and, optionally, the matrices *T* and *Z* from the Schur decomposition $H = Z^*T^*Z^T$, where *T* is an upper quasi-triangular matrix (the Schur form), and *Z* is the orthogonal matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal matrix Q so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal matrix Q : $A = Q^*H^*Q^T = (QZ)^*T^*(QZ)^T$.

Input Parameters

<i>job</i>	(global) CHARACTER*1 = 'E': compute eigenvalues only; = 'S': compute eigenvalues and the Schur form T .
<i>compz</i>	(global) CHARACTER*1 = 'N': no Schur vectors are computed; = 'I': z is initialized to the unit matrix and the matrix Z of Schur vectors of h is returned; = 'V': z must contain an orthogonal matrix Q on entry, and the product Q^*Z is returned.
<i>n</i>	(global) INTEGER The order of the Hessenberg matrix h . $n \geq 0$.
<i>ilo, ihi</i>	(global) INTEGER It is assumed that h is already upper triangular in rows and columns 1: $ilo-1$ and $ihi+1:n$. <i>ilo</i> and <i>ihi</i> are normally set by a previous call to p?gebal , and then passed to p?gehrd when the matrix output by p?gebal is reduced to Hessenberg form. Otherwise <i>ilo</i> and <i>ihi</i> should be set to 1 and <i>n</i> respectively. If $n > 0$, then $1 \leq ilo \leq ihi \leq n$. If $n = 0$, then <i>ilo</i> = 1 and <i>ihi</i> = 0.
<i>h</i>	REAL for pshseqr DOUBLE PRECISION for pdhseqr (global) array, dimension (<i>desch</i> (<i>lld_</i>), <i>LOC_c(n)</i>) The upper Hessenberg matrix H .
<i>desch</i>	(global and local) INTEGER array of dimension <i>dlen_</i> The array descriptor for the distributed matrix h .
<i>z</i>	REAL for pshseqr DOUBLE PRECISION for pdhseqr (global) array If <i>compz</i> = 'V', on entry z must contain the current matrix Z of accumulated transformations from, for example, p?gehrd . If <i>compz</i> = 'I', on entry z need not be set.
<i>descz</i>	(global and local) INTEGER array of dimension <i>dlen_</i> The array descriptor for the distributed matrix z .
<i>work</i>	REAL for pshseqr

DOUBLE PRECISION for pdhseqr
 (local workspace) array, dimension(*lwork*)

lwork (local) INTEGER
 The length of the workspace array *work*.

iwork (local workspace) INTEGER array, dimension (*liwork*)

liwork (local) INTEGER
 The length of the workspace array *iwork*.

OUTPUT Parameters

h If *job* = 'S', *h* is upper quasi-triangular in rows and columns *ilo:ichi*, with 1-by-1 and 2-by-2 blocks on the main diagonal. The 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with $h(i,i) = h(i+1,i+1)$ and $h(i+1,i)*h(i,i+1) < 0$. If *info* = 0 and *job* = 'E', the contents of *h* are unspecified on exit.

wr, wi REAL for pshseqr
 DOUBLE PRECISION for pdhseqr
 (global) array, dimension (*n*)
 The real and imaginary parts, respectively, of the computed eigenvalues *ilo* to *ichi* are stored in the corresponding elements of *wr* and *wi*. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of *wr* and *wi*, say the *i*-th and *(i+1)*-th, with *wi(i) > 0* and *wi(i+1) < 0*. If *job* = 'S', the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in *h*.

z REAL for pshseqr
 DOUBLE PRECISION for pdhseqr
 (global) array
z is updated; transformations are applied only to the submatrix *z(ilo:ichi,ilo:ichi)*.
 If *compz* = 'N', *z* is not referenced.
 If *compz* = 'I' and *info* = 0, *z* contains the orthogonal matrix *Z* of the Schur vectors of *h*.

info INTEGER
 = 0: successful exit
 < 0: if *info* = *-i*, the *i*-th argument had an illegal value (see also below for -7777 and -8888).
 > 0: if *info* = *i*, p?hseqr failed to compute all of the eigenvalues. Elements 1:*ilo*-1 and *i+1:n* of *wr* and *wi* contain those eigenvalues which have been successfully computed. (Failures are rare.)

If $info > 0$ and $job = 'E'$, then on exit, the remaining un converged eigenvalues are the eigen- values of the upper Hessenberg matrix rows and columns i_0 through $info$ of the final, output value of h .

If $info > 0$ and $job = 'S'$, then on exit (*) (initial value of H)* $U = U$ *(final value of H) where U is an orthogonal matrix. The final value of H is upper Hessenberg and quasi-triangular in rows and columns $info+1$ through ihi .

If $info > 0$ and $compz = 'V'$, then on exit (final value of Z) = (initial value of Z)* U where U is the orthogonal matrix in (*) (regardless of the value of job .)

If $info > 0$ and $compz = 'I'$, then on exit (final value of Z) = U where U is the orthogonal matrix in (*) (regardless of the value of job .)

If $info > 0$ and $compz = 'N'$, then z is not accessed.

= -7777: `p?laqr0` failed to converge and `p?laqr1` was called instead.

= -8888: `p?laqr1` failed to converge and `p?laqr0` was called instead.

Singular Value Decomposition

This section describes ScALAPACK routines for computing the singular value decomposition (SVD) of a general m -by- n matrix A (see "Singular Value Decomposition" in LAPACK chapter).

To find the SVD of a general matrix A , this matrix is first reduced to a bidiagonal matrix B by a unitary (orthogonal) transformation, and then SVD of the bidiagonal matrix is computed. Note that the SVD of B is computed using the LAPACK routine `?bdsqr`.

[Table "Computational Routines for Singular Value Decomposition \(SVD\)"](#) lists ScALAPACK computational routines for performing this decomposition.

Computational Routines for Singular Value Decomposition (SVD)

Operation	General matrix	Orthogonal/unitary matrix
Reduce A to a bidiagonal matrix	<code>p?gebrd</code>	
Multiply matrix after reduction		<code>p?ormbr/p?unmbr</code>

`p?gebrd`

Reduces a general matrix to bidiagonal form.

Syntax

Fortran:

```
call psgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pdgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pcgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pzgebrd(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

C:

```
void psgebrd (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *d , float *e , float *tauq , float *taup , float *work , MKL_INT
*lwork , MKL_INT *info );
void pdgebrd (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *d , double *e , double *tauq , double *taup , double *work ,
MKL_INT *lwork , MKL_INT *info );
```

```

void pcgebrd (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *d , float *e , MKL_Complex8 *tauq , MKL_Complex8 *taup ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzgebrd (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *d , double *e , MKL_Complex16 *tauq , MKL_Complex16 *taup ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The p?gebrd routine reduces a real/complex general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form B by an orthogonal/unitary transformation:

$$Q' \cdot \text{sub}(A) \cdot P = B.$$

If $m \geq n$, B is upper bidiagonal; if $m < n$, B is lower bidiagonal.

Input Parameters

m	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(A)$ ($m \geq 0$).
n	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
a	(local) REAL for psgebrd DOUBLE PRECISION for pdgebrd COMPLEX for pcgebrd DOUBLE COMPLEX for pzgebrd. Real pointer into the local memory to an array of dimension ($l1d_a$, $\text{LOCc}(ja+n-1)$). On entry, this array contains the distributed matrix $\text{sub}(A)$.
ia, ja	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.
$desca$	(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .
$work$	(local) REAL for psgebrd DOUBLE PRECISION for pdgebrd COMPLEX for pcgebrd DOUBLE COMPLEX for pzgebrd. Workspace array of dimension $lwork$.
$lwork$	(local or global) INTEGER, dimension of $work$, must be at least: $lwork \geq nb * (mpa0 + nqa0 + 1) + nqa0$

where $nb = mb_a = nb_a$,
 $iroffa = \text{mod}(ia-1, nb)$,
 $icoffa = \text{mod}(ja-1, nb)$,
 $iarow = \text{indxg2p}(ia, nb, MYROW, rsrc_a, NPROW)$,
 $iacol = \text{indxg2p}(ja, nb, MYCOL, csrc_a, NPCOL)$,
 $npa0 = \text{numroc}(m + iroffa, nb, MYROW, iarow, NPROW)$,
 $nqa0 = \text{numroc}(n + icoffa, nb, MYCOL, iacol, NPCOL)$.

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a

On exit, if $m \geq n$, the diagonal and the first superdiagonal of sub(A) are overwritten with the upper bidiagonal matrix B ; the elements below the diagonal, with the array τ_{aq} , represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array τ_{ap} , represent the orthogonal matrix P as a product of elementary reflectors. If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B ; the elements below the first subdiagonal, with the array τ_{aq} , represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array τ_{ap} , represent the orthogonal matrix P as a product of elementary reflectors. See *Application Notes* below.

d

(local)

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors. Array, size $LOCc(ja + \min(m, n) - 1)$ if $m \geq n$; $LOCr(ia + \min(m, n) - 1)$ otherwise. The distributed diagonal elements of the bidiagonal matrix B : $d(i) = a(i, i)$.

d is tied to the distributed matrix A .

e

(local)

REAL for single-precision flavors

DOUBLE PRECISION for double-precision flavors. Array, size $LOCr(ia + \min(m, n) - 1)$ if $m \geq n$; $LOCc(ja + \min(m, n) - 2)$ otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix B :

If $m \geq n$, $e(i) = a(i, i+1)$ for $i = 1, 2, \dots, n-1$; if $m < n$, $e(i) = a(i+1, i)$ for $i = 1, 2, \dots, m-1$. e is tied to the distributed matrix A .

τ_{aq}, τ_{ap}

(local)

REAL for psgebrd

DOUBLE PRECISION for pdgebrd

COMPLEX for pcgebrd

DOUBLE COMPLEX for pzgebrd.

Arrays, size $LOCc(ja+\min(m,n)-1)$ for τ_{aq} and $LOCr(ia+\min(m,n)-1)$ for τ_{ap} . Contain the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrices Q and P , respectively. τ_{aq} and τ_{ap} are tied to the distributed matrix A . See Application Notes below.

$work(1)$

On exit $work(1)$ contains the minimum value of $lwork$ required for optimum performance.

$info$

(global) INTEGER.

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrices Q and P are represented as products of elementary reflectors:

If $m \geq n$,

$Q = H(1)*H(2)*\dots*H(n)$, and $P = G(1)*G(2)*\dots*G(n-1)$.

Each $H(i)$ and $G(i)$ has the form:

$H(i) = i - \tau_{aq} * v * v'$ and $G(i) = i - \tau_{ap} * u * u'$

where τ_{aq} and τ_{ap} are real/complex scalars, and v and u are real/complex vectors;

$v(1:i-1) = 0$, $v(i) = 1$, and $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$;

$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+2:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;

τ_{aq} is stored in $\tau_{aq}(ja+i-1)$ and τ_{ap} in $\tau_{ap}(ia+i-1)$.

If $m < n$,

$Q = H(1)*H(2)*\dots*H(m-1)$, and $P = G(1)*G(2)*\dots*G(m)$

Each $H(i)$ and $G(i)$ has the form:

$H(i) = i - \tau_{aq} * v * v'$ and $G(i) = i - \tau_{ap} * u * u'$

here τ_{aq} and τ_{ap} are real/complex scalars, and v and u are real/complex vectors;

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$; $u(1:i-1) = 0$, $u(i) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;

τ_{aq} is stored in $\tau_{aq}(ja+i-1)$ and τ_{ap} in $\tau_{ap}(ia+i-1)$.

The contents of $\text{sub}(A)$ on exit are illustrated by the following examples:

$m = 6$ and $n = 5$ ($m > n$):

$$\begin{bmatrix} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{bmatrix}$$

$m = 5$ and $n = 6$ ($m < n$):

$$\begin{bmatrix} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{bmatrix}$$

where d and e denote diagonal and off-diagonal elements of B , vi denotes an element of the vector defining $H(i)$, and ui an element of the vector defining $G(i)$.

p?ormbr

Multiples a general matrix by one of the orthogonal matrices from a reduction to bidiagonal form determined by p?gebrd.

Syntax

Fortran:

```
call psormbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

```
call pdormbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc,
work, lwork, info)
```

C:

```
void psormbr (char *vect , char *side , char *trans , MKL_INT *m , MKL_INT *n ,
MKL_INT *k , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau ,
float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork ,
MKL_INT *info );

void pdormbr (char *vect , char *side , char *trans , MKL_INT *m , MKL_INT *n ,
MKL_INT *k , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau ,
double *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

If $\text{vect} = 'Q'$, the p?ormbr routine overwrites the general real distributed m -by- n matrix $\text{sub}(C) = C(c:ic + m - 1, jc:jc + n - 1)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$Q \text{ sub}(C)$	$\text{sub}(C) Q$
<i>trans</i> = 'T':	$Q^T \text{ sub}(C)$	$\text{sub}(C) Q^T$

If *vect* = 'P', the routine overwrites $\text{sub}(C)$ with

	<i>side</i> = 'L'	<i>side</i> = 'R'
<i>trans</i> = 'N':	$P \text{ sub}(C)$	$\text{sub}(C) P$
<i>trans</i> = 'T':	$P^T \text{ sub}(C)$	$\text{sub}(C) P^T$

Here Q and P^T are the orthogonal distributed matrices determined by [p?gebrd](#) when reducing a real distributed matrix $A(ia:*, ja:*)$ to bidiagonal form: $A(ia:*, ja:*) = Q^* B^* P^T$. Q and P^T are defined as products of elementary reflectors $H(i)$ and $G(i)$ respectively.

Let $nq = m$ if *side* = 'L' and $nq = n$ if *side* = 'R'. Thus nq is the order of the orthogonal matrix Q or P^T that is applied.

If *vect* = 'Q', $A(ia:*, ja:*)$ is assumed to have been an nq -by- k matrix:

If $nq \geq k$, $Q = H(1) H(2) \dots H(k)$;

If $nq < k$, $Q = H(1) H(2) \dots H(nq-1)$.

If *vect* = 'P', $A(ia:*, ja:*)$ is assumed to have been a k -by- nq matrix:

If $k < nq$, $P = G(1) G(2) \dots G(k)$;

If $k \geq nq$, $P = G(1) G(2) \dots G(nq-1)$.

Input Parameters

<i>vect</i>	(global) CHARACTER. If <i>vect</i> = 'Q', then Q or Q^T is applied. If <i>vect</i> = 'P', then P or P^T is applied.
<i>side</i>	(global) CHARACTER. If <i>side</i> = 'L', then Q or Q^T , P or P^T is applied from the left. If <i>side</i> = 'R', then Q or Q^T , P or P^T is applied from the right.
<i>trans</i>	(global) CHARACTER. If <i>trans</i> = 'N', no transpose, Q or P is applied. If <i>trans</i> = 'T', then Q^T or P^T is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix $\text{sub}(C)$.
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix $\text{sub}(C)$.
<i>k</i>	(global) INTEGER. If <i>vect</i> = 'Q', the number of columns in the original distributed matrix reduced by p?gebrd ; If <i>vect</i> = 'P', the number of rows in the original distributed matrix reduced by p?gebrd . Constraints: $k \geq 0$.

<i>a</i>	(local) REAL for psormbr DOUBLE PRECISION for pdormbr. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja</i> +min(<i>nq</i> , <i>k</i>)-1)) If <i>vect</i> ='Q', and (<i>lld_a</i> , <i>LOCc(ja+nq-1)</i>) If <i>vect</i> = 'P'. <i>nq</i> = <i>m</i> if <i>side</i> = 'L', and <i>nq</i> = <i>n</i> otherwise. The vectors which define the elementary reflectors <i>H(i)</i> and <i>G(i)</i> , whose products determine the matrices <i>Q</i> and <i>P</i> , as returned by p?gebrd. If <i>vect</i> = 'Q', <i>lld_a</i> ≥max(1, <i>LOCr(ia+nq-1)</i>); If <i>vect</i> = 'P', <i>lld_a</i> ≥max(1, <i>LOCr(ia+min(nq, k)-1)</i>).
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) REAL for psormbr DOUBLE PRECISION for pdormbr. Array, size <i>LOCc(ja+min(nq, k)-1)</i> , if <i>vect</i> = 'Q', and <i>LOCr(ia+min(nq, k)-1)</i> , if <i>vect</i> = 'P'. <i>tau(i)</i> must contain the scalar factor of the elementary reflector <i>H(i)</i> or <i>G(i)</i> , which determines <i>Q</i> or <i>P</i> , as returned by pdgebrd in its array argument <i>tauq</i> or <i>taup</i> . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) REAL for psormbr DOUBLE PRECISION for pdormbr Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(jc</i> + <i>n</i> -1)). Contains the local pieces of the distributed matrix sub (<i>C</i>).
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix <i>C</i> , respectively.
<i>descC</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local) REAL for psormbr DOUBLE PRECISION for pdormbr.

Workspace array of dimension *lwork*.

lwork

(local or global) INTEGER, dimension of *work*, must be at least:

```

If side = 'L'

nq = m;

if ((vect = 'Q' and nq≥k) or (vect is not equal to 'Q' and nq>k)),
iaa=ia; jaa=ja; mi=m; ni=n; icc=ic; jcc=jc;

else

iaa= ia+1; jaa=ja; mi=m-1; ni=n; icc=ic+1; jcc= jc;

end if

else

If side = 'R', nq = n;

if((vect = 'Q' and nq≥k) or (vect is not equal to 'Q' and
nq>k)),
iaa=ia; jaa=ja; mi=m; ni=n; icc=ic; jcc=jc;

else

iaa= ia; jaa= ja+1; mi= m; ni= n-1; icc= ic; jcc= jc+1;

end if

end if

If vect = 'Q',

If side = 'L', lwork≥max((nb_a*(nb_a-1))/2, (nqc0 + mpc0)*nb_a) +
nb_a * nb_a

else if side = 'R',

lwork≥max((nb_a*(nb_a-1))/2, (nqc0 + max(npao +
numroc(numroc(ni+icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0,
lcmq), mpc0))*nb_a) + nb_a*nb_a

end if

else if vect is not equal to 'Q', if side = 'L',

lwork≥max((mb_a*(mb_a-1))/2, (mpc0 + max(mqa0 +
numroc(numroc(mi+iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0,
lcmp), nqc0))*mb_a) + mb_a*mb_a

else if side = 'R',

lwork≥max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a) + mb_a*mb_a

end if

end if

where lcm = lcm/NPROW, lcmq = lcm/NPCOL, with lcm =
ilcm(NPROW, NPCOL),

iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),

```

```

iacol = indxg2p(jaa, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(mi+icooffa, nb_a, MYCOL, iacol, NPCOL),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(icc-1, mb_c),
icooffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(ni+icooffc, nb_c, MYCOL, iccol, NPCOL),
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW
and NPCOL can be determined by calling the subroutine blacs_gridinfo.
If lwork = -1, then lwork is global input and a workspace query is
assumed; the routine only calculates the minimum and optimal size for all
work arrays. Each of these values is returned in the first entry of the
corresponding work array, and no error message is issued by pxerbla.

```

Output Parameters

<i>c</i>	On exit, if <i>vect</i> ='Q', <i>sub(C)</i> is overwritten by $Q^*sub(C)$, or $Q'^*sub(C)$, or $sub(C)*Q'$, or $sub(C)*Q$; if <i>vect</i> ='P', <i>sub(C)</i> is overwritten by $P^*sub(C)$, or $P'^*sub(C)$, or $sub(C)*P$, or $sub(C)*P'$.
<i>work(1)</i>	On exit <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	(global) INTEGER. = 0: the execution is successful. < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (i* 100+j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?unmbr

Multiples a general matrix by one of the unitary transformation matrices from a reduction to bidiagonal form determined by p?gebrd.

Syntax

Fortran:

```

call pcunmbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, desc,
work, lwork, info)

call pzunmbr(vect, side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, desc,
work, lwork, info)

```

C:

```

void pcunmbr (char *vect , char *side , char *trans , MKL_INT *m , MKL_INT *n ,
MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_Complex8 *tau , MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzunmbr (char *vect , char *side , char *trans , MKL_INT *m , MKL_INT *n ,
MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
MKL_Complex16 *tau , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

If $\text{vect} = 'Q'$, the `p?unmbr` routine overwrites the general complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

	$\text{side} = 'L'$	$\text{side} = 'R'$
$\text{trans} = 'N':$	$Q^*\text{sub}(C)$	$\text{sub}(C)*Q$
$\text{trans} = 'C':$	$Q^H*\text{sub}(C)$	$\text{sub}(C)*Q^H$

If $\text{vect} = 'P'$, the routine overwrites $\text{sub}(C)$ with

	$\text{side} = 'L'$	$\text{side} = 'R'$
$\text{trans} = 'N':$	$P^*\text{sub}(C)$	$\text{sub}(C)*P$
$\text{trans} = 'C':$	$P^H*\text{sub}(C)$	$\text{sub}(C)*P^H$

Here Q and P^H are the unitary distributed matrices determined by `p?gebrd` when reducing a complex distributed matrix $A(ia:*, ja:*)$ to bidiagonal form: $A(ia:*, ja:*) = Q^*B^*P^H$.

Q and P^H are defined as products of elementary reflectors $H(i)$ and $G(i)$ respectively.

Let $nq = m$ if $\text{side} = 'L'$ and $nq = n$ if $\text{side} = 'R'$. Thus nq is the order of the unitary matrix Q or P^H that is applied.

If $\text{vect} = 'Q'$, $A(ia:*, ja:*)$ is assumed to have been an nq -by- k matrix:

If $nq \geq k$, $Q = H(1) H(2) \dots H(k)$;

If $nq < k$, $Q = H(1) H(2) \dots H(nq-1)$.

If $\text{vect} = 'P'$, $A(ia:*, ja:*)$ is assumed to have been a k -by- nq matrix:

If $k < nq$, $P = G(1) G(2) \dots G(k)$;

If $k \geq nq$, $P = G(1) G(2) \dots G(nq-1)$.

Input Parameters

<code>vect</code>	(global) CHARACTER.
	If $\text{vect} = 'Q'$, then Q or Q^H is applied.
	If $\text{vect} = 'P'$, then P or P^H is applied.
<code>side</code>	(global) CHARACTER.
	If $\text{side} = 'L'$, then Q or Q^H , P or P^H is applied from the left.
	If $\text{side} = 'R'$, then Q or Q^H , P or P^H is applied from the right.

<i>trans</i>	(global) CHARACTER. If <i>trans</i> = 'N', no transpose, <i>Q</i> or <i>P</i> is applied. If <i>trans</i> = 'C', conjugate transpose, Q^H or P^H is applied.
<i>m</i>	(global) INTEGER. The number of rows in the distributed matrix sub (<i>C</i>) $n \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns in the distributed matrix sub (<i>C</i>) $n \geq 0$.
<i>k</i>	(global) INTEGER. If <i>vect</i> = 'Q', the number of columns in the original distributed matrix reduced by p?gebrd; If <i>vect</i> = 'P', the number of rows in the original distributed matrix reduced by p?gebrd. Constraints: $k \geq 0$.
<i>a</i>	(local) COMPLEX for psormbr DOUBLE COMPLEX for pdormbr. Pointer into the local memory to an array of dimension (<i>lld_a</i> , $LOCc(ja + \min(nq, k) - 1)$) if <i>vect</i> ='Q', and (<i>lld_a</i> , $LOCc(ja + nq - 1)$) if <i>vect</i> = 'P'. $nq = m$ if <i>side</i> = 'L', and $nq = n$ otherwise. The vectors which define the elementary reflectors <i>H</i> (<i>i</i>) and <i>G</i> (<i>i</i>), whose products determine the matrices <i>Q</i> and <i>P</i> , as returned by p?gebrd. If <i>vect</i> = 'Q', $lld_a \geq \max(1, LOCr(ia + nq - 1))$; If <i>vect</i> = 'P', $lld_a \geq \max(1, LOCr(ia + \min(nq, k) - 1))$.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>tau</i>	(local) COMPLEX for pcunmbr DOUBLE COMPLEX for pzunmbr. Array, size $LOCc(ja + \min(nq, k) - 1)$, if <i>vect</i> = 'Q', and $LOCr(ia + \min(nq, k) - 1)$, if <i>vect</i> = 'P'. <i>tau</i> (<i>i</i>) must contain the scalar factor of the elementary reflector <i>H</i> (<i>i</i>) or <i>G</i> (<i>i</i>), which determines <i>Q</i> or <i>P</i> , as returned by p?gebrd in its array argument <i>tauq</i> or <i>taup</i> . <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>c</i>	(local) COMPLEX for pcunmbr

DOUBLE COMPLEX for pzunmbr
 Pointer into the local memory to an array of dimension (*lld_a*, *LOCc* (*jc* +*n*-1)).
 Contains the local pieces of the distributed matrix sub (*C*).
ic, jc
 (global) INTEGER. The row and column indices in the global array *c* indicating the first row and the first column of the submatrix *C*, respectively.
descC
 (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *C*.
work
 (local)
 COMPLEX for pcunmbr
 DOUBLE COMPLEX for pzunmbr.
 Workspace array of dimension *lwork*.
lwork
 (local or global) INTEGER, dimension of *work*, must be at least:
If *side* = 'L'
nq = *m*;
if ((*vect* = 'Q' and *nq* ≥ *k*) or (*vect* is not equal to 'Q' and *nq* > *k*)), *iaa*= *ia*; *jaa*= *ja*; *mi*= *m*; *ni*= *n*; *icc*= *ic*; *jcc*= *jc*;
else
iaa= *ia*+1; *jaa*= *ja*; *mi*= *m*-1; *ni*= *n*; *icc*= *ic*+1; *jcc*= *jc*;
end if
else
If *side* = 'R', *nq* = *n*;
if ((*vect* = 'Q' and *nq* ≥ *k*) or (*vect* is not equal to 'Q' and *nq* ≥ *k*)),
iaa= *ia*; *jaa*= *ja*; *mi*= *m*; *ni*= *n*; *icc*= *ic*; *jcc*= *jc*;
else
iaa= *ia*; *jaa*= *ja*+1; *mi*= *m*; *ni*= *n*-1; *icc*= *ic*; *jcc*= *jc*+1;
end if
end if
If *vect* = 'Q',
If *side* = 'L', *lwork* ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc0*+*mpc0*)**nb_a*) + *nb_a***nb_a*
else if *side* = 'R',
lwork ≥ max((*nb_a**(*nb_a*-1))/2, (*nqc0* + max(*npa0*+*numroc*(*numroc*(*ni*+*icoffc*, *nb_a*, 0, 0, *NPCOL*), *nb_a*, 0, 0, *lcmq*), *mpc0*))**nb_a*) + *nb_a***nb_a*
end if
else if *vect* is not equal to 'Q',

```

if side = 'L',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 +
max(mqa0+numroc(numroc(mi+iroffc, mb_a, 0, 0, NPROW), mb_a,
0, 0, lcmp), nqc0))*mb_a) + mb_a*mb_a
else if side = 'R',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a) +
mb_a*mb_a
end if
end if
where lcmq = lcm/NPROW, lcmq = lcm/NPCOL, with lcm = ilcm(
NPROW, NPCOL),
iroffa = mod(iaa-1, mb_a),
icooffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
iacol = indxg2p(jaa, nb_a, MYCOL, csrc_a, NPCOL),
mqa0 = numroc(mi+icooffa, nb_a, MYCOL, iacol, NPCOL),
nqa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(jcc-1, mb_c),
icooffc = mod(jcc-1, nb_c),
icrow = indxg2p(jcc, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(ni+icooffc, nb_c, MYCOL, iccol, NPCOL),
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW
and NPCOL can be determined by calling the subroutine blacs_gridinfo.
```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c

On exit, if *vect*='Q', *sub(C)* is overwritten by $Q^* \text{sub}(C)$, or $Q'^* \text{sub}(C)$, or $\text{sub}(C) * Q'$, or $\text{sub}(C) * Q$; if *vect*='P', *sub(C)* is overwritten by $P^* \text{sub}(C)$, or $P'^* \text{sub}(C)$, or $\text{sub}(C) * P$, or $\text{sub}(C) * P'$.

work(1)

On exit *work(1)* contains the minimum value of *lwork* required for optimum performance.

info

(global) INTEGER.

= 0: the execution is successful.

< 0 : if the i -th argument is an array and the j -entry had an illegal value, then $\text{info} = - (i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.

Generalized Symmetric-Definite Eigen Problems

This section describes ScalAPACK routines that allow you to reduce the *generalized symmetric-definite eigenvalue problems* (see [Generalized Symmetric-Definite Eigenvalue Problems](#) in LAPACK chapters) to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling ScalAPACK routines described earlier in this chapter (see [Symmetric Eigenproblems](#)).

[Table "Computational Routines for Reducing Generalized Eigenproblems to Standard Problems"](#) lists these routines.

Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

Operation	Real symmetric matrices	Complex Hermitian matrices
Reduce to standard problems	p?sygst	p?hegst

p?sygst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

Fortran:

```
call pssygst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
call pdsygst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
```

C:

```
void pssygst (MKL_INT *ibtype , char *uplo , MKL_INT *n , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
float *scale , MKL_INT *info );
void pdsygst (MKL_INT *ibtype , char *uplo , MKL_INT *n , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , double *scale , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?sygst` routine reduces real symmetric-definite generalized eigenproblems to the standard form.

In the following $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If $ibtype = 1$, the problem is

$\text{sub}(A) * x = \lambda * \text{sub}(B) * x$,

and $\text{sub}(A)$ is overwritten by $\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$.

If $ibtype = 2$ or 3 , the problem is

$\text{sub}(A) * \text{sub}(B) * x = \lambda * x$, or $\text{sub}(B) * \text{sub}(A) * x = \lambda * x$,

and $\text{sub}(A)$ is overwritten by $U * \text{sub}(A) * U^T$, or $L^T * \text{sub}(A) * L$.

$\text{sub}(B)$ must have been previously factorized as $U^T * U$ or $L * L^T$ by p?potrf .

Input Parameters

<i>ibtype</i>	(global) INTEGER. Must be 1 or 2 or 3. If <i>ibtype</i> = 1, compute $\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$; If <i>ibtype</i> = 2 or 3, compute $U * \text{sub}(A) * U^T$, or $L^T * \text{sub}(A) * L$.
<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factored as $U^T * U$. If <i>uplo</i> = 'L', the lower triangle of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factored as $L * L^T$.
<i>n</i>	(global) INTEGER. The order of the matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).
<i>a</i>	(local) REAL for pssygst DOUBLE PRECISION for pdsgst. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). On entry, the array contains the local pieces of the n -by- n symmetric distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = 'L', the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) REAL for pssygst DOUBLE PRECISION for pdsgst. Pointer into the local memory to an array of dimension (<i>lld_b</i> , <i>LOCc(jb+n-1)</i>). On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of $\text{sub}(B)$ as returned by p?potrf .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>b</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, the transformed matrix, stored in the same format as $\text{sub}(A)$.
<i>scale</i>	(global) REAL for pssygst DOUBLE PRECISION for pdsygst. Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, <i>scale</i> is always returned as 1.0, it is returned here to allow for future enhancement.
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?hegst

Reduces a Hermitian positive-definite generalized eigenvalue problem to the standard form.

Syntax

Fortran:

```
call pcgdst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
call pzgdst(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, scale, info)
```

C:

```
void pcgdst (MKL_INT *ibtype , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , float *scale , MKL_INT *info );
void pzgdst (MKL_INT *ibtype , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , double *scale , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?hegst routine reduces complex Hermitian positive-definite generalized eigenproblems to the standard form.

In the following $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If *ibtype* = 1, the problem is

$\text{sub}(A) * x = \lambda * \text{sub}(B) * x$,

and $\text{sub}(A)$ is overwritten by $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$.

If *ibtype* = 2 or 3, the problem is

$\text{sub}(A) * \text{sub}(B) * x = \lambda * x$, or $\text{sub}(B) * \text{sub}(A) * x = \lambda * x$,

and $\text{sub}(A)$ is overwritten by $U^* \text{sub}(A) * U^H$, or $L^H * \text{sub}(A) * L$.

$\text{sub}(B)$ must have been previously factorized as $U^H * U$ or $L * L^H$ by [p?potrf](#).

Input Parameters

<i>ibtype</i>	(global) INTEGER. Must be 1 or 2 or 3. If <i>itype</i> = 1, compute $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$; If <i>itype</i> = 2 or 3, compute $U^* \text{sub}(A) * U^H$, or $L^H * \text{sub}(A) * L$.
<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'. If <i>uplo</i> = 'U', the upper triangle of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factored as $U^H * U$. If <i>uplo</i> = 'L', the lower triangle of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factored as $L * L^H$.
<i>n</i>	(global) INTEGER. The order of the matrices $\text{sub}(A)$ and $\text{sub}(B)$ ($n \geq 0$).
<i>a</i>	(local) COMPLEX for pchegst DOUBLE COMPLEX for pzhegst . Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCC(ja</i> + <i>n</i> -1)). On entry, the array contains the local pieces of the <i>n</i> -by- <i>n</i> Hermitian distributed matrix $\text{sub}(A)$. If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>b</i>	(local) COMPLEX for pchegst DOUBLE COMPLEX for pzhegst . Pointer into the local memory to an array of dimension (<i>lld_b</i> , <i>LOCC(jb</i> + <i>n</i> -1)). On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of $\text{sub}(B)$ as returned by p?potrf .
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>b</i> indicating the first row and the first column of the submatrix <i>B</i> , respectively.
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> .

Output Parameters

<i>a</i>	On exit, if <i>info</i> = 0, the transformed matrix, stored in the same format as sub(<i>A</i>).
<i>scale</i>	(global) REAL for pchegst DOUBLE PRECISION for pzhegst. Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, <i>scale</i> is always returned as 1.0, it is returned here to allow for future enhancement.
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0, if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> 100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Driver Routines

Table "ScaLAPACK Driver Routines" lists ScaLAPACK driver routines available for solving systems of linear equations, linear least-squares problems, standard eigenvalue and singular value problems, and generalized symmetric definite eigenproblems.

ScaLAPACK Driver Routines

Type of Problem	Matrix type, storage scheme	Driver
Linear equations	general (partial pivoting)	p?gesv (simple driver) / p?gesvx (expert driver)
	general band (partial pivoting)	p?gbsv (simple driver)
	general band (no pivoting)	p?dbsv (simple driver)
	general tridiagonal (no pivoting)	p?dtsv (simple driver)
	symmetric/Hermitian positive-definite	p?posv (simple driver) / p?posvx (expert driver)
	symmetric/Hermitian positive-definite, band	p?pbsv (simple driver)
	symmetric/Hermitian positive-definite, tridiagonal	p?ptsv (simple driver)
Linear least squares problem	general <i>m</i> -by- <i>n</i>	p?gels
Symmetric eigenvalue problem	symmetric/Hermitian	p?syev / p?heev (simple driver); p?syevd / p?heevd (simple driver with a divide and conquer algorithm); p?syevx / p?heevx (expert driver); p?syevr / p?heevr (simple driver with MRRR algorithm)
Singular value decomposition	general <i>m</i> -by- <i>n</i>	p?gesvd
Generalized symmetric definite eigenvalue problem	symmetric/Hermitian, one matrix also positive-definite	p?sygvx / p?hegvx (expert driver)

[p?gesv](#)

Computes the solution to the system of linear equations with a square distributed matrix and multiple right-hand sides.

Syntax

Fortran:

```
call psgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pdgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pcgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
call pzgesv(n, nrhs, a, ia, ja, desca, ipiv, b, ib, jb, descb, info)
```

C:

```
void psgesv (MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
void pdgesv (MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
void pcgesv (MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
void pzgesv (MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?gesv` routine computes the solution to a real or complex system of linear equations $\text{sub}(A) * X = \text{sub}(B)$, where $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ is an n -by- n distributed matrix and X and $\text{sub}(B) = B(ib:ib+n-1, jb:jb+nrhs-1)$ are n -by- $nrhs$ distributed matrices.

The *LU* decomposition with partial pivoting and row interchanges is used to factor $\text{sub}(A)$ as $\text{sub}(A) = P * L * U$, where P is a permutation matrix, L is unit lower triangular, and U is upper triangular. L and U are stored in $\text{sub}(A)$. The factored form of $\text{sub}(A)$ is then used to solve the system of equations $\text{sub}(A) * X = \text{sub}(B)$.

Input Parameters

<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right hand sides, that is, the number of columns of the distributed submatrices B and X ($nrhs \geq 0$).
<i>a, b</i>	(local) REAL for psgesv DOUBLE PRECISION for pdgesv COMPLEX for pcgesv DOUBLE COMPLEX for pzgesv. Pointers into the local memory to arrays of local dimension $a(lld_a, LOCc(ja+n-1))$ and $b(lld_b, LOCc(jb+nrhs-1))$, respectively.

On entry, the array a contains the local pieces of the n -by- n distributed matrix sub(A) to be factored.

On entry, the array b contains the right hand side distributed matrix sub(B).

ia, ja (global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of sub(A), respectively.

$desc_a$ (global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .

ib, jb (global) INTEGER. The row and column indices in the global array B indicating the first row and the first column of sub(B), respectively.

$desc_b$ (global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix B .

Output Parameters

a Overwritten by the factors L and U from the factorization sub(A) = $P*L*U$; the unit diagonal elements of L are not stored .

b Overwritten by the solution distributed matrix X .

$ipiv$ (local) INTEGER array.

The dimension of $ipiv$ is ($LOCr(m_a) + mb_a$) . This array contains the pivoting information. The (local) row i of the matrix was interchanged with the (global) row $ipiv(i)$.

This array is tied to the distributed matrix A .

$info$ (global) INTEGER. If $info=0$, the execution is successful.

$info < 0$:

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$; if the i -th argument is a scalar and had an illegal value, then $info = -i$.

$info > 0$:

If $info = k$, $U(ia+k-1,ja+k-1)$ is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution could not be computed.

p?gesvx

Uses the LU factorization to compute the solution to the system of linear equations with a square matrix A and multiple right-hand sides, and provides error bounds on the solution.

Syntax

Fortran:

```
call psgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descraf, ipiv,
equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork,
iwork, liwork, info)
```

```
call pdgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descraf, ipiv,
equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork,
iwork, liwork, info)
```

```
call pcgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descraf, ipiv,
equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork,
rwork, lrwork, info)
```

```
call pzgesvx(fact, trans, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descraf, ipiv,
equed, r, c, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork,
rwork, lrwork, info)
```

C:

```
void psgesvx (char *fact , char *trans , MKL_INT *n , MKL_INT *nrhs , float *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *iaf , MKL_INT *jaf ,
MKL_INT *descraf , MKL_INT *ipiv , char *equed , float *r , float *c , float *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , float *rcond , float *ferr , float *berr , float *work , MKL_INT
*lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pdgesvx (char *fact , char *trans , MKL_INT *n , MKL_INT *nrhs , double *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *af , MKL_INT *iaf , MKL_INT
*jaf , MKL_INT *descraf , MKL_INT *ipiv , char *equed , double *r , double *c , double
*b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , double *rcond , double *ferr , double *berr , double *work ,
MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pcgesvx (char *fact , char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *iaf , MKL_INT
*jaf , MKL_INT *descraf , MKL_INT *ipiv , char *equed , float *r , float *c ,
MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *rcond , float *ferr , float
*berr , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT
*info );

void pzgesvx (char *fact , char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *iaf ,
MKL_INT *jaf , MKL_INT *descraf , MKL_INT *ipiv , char *equed , double *r , double *c ,
MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *rcond , double *ferr , double
*berr , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT *lrwork ,
MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?gesvx` routine uses the *LU* factorization to compute the solution to a real or complex system of linear equations $AX = B$, where A denotes the n -by- n submatrix $A(ia:ia+n-1, ja:ja+n-1)$, B denotes the n -by- $nrhs$ submatrix $B(ib:ib+n-1, jb:jb+nrhs-1)$ and X denotes the n -by- $nrhs$ submatrix $X(ix:ix+n-1, jx:jx+nrhs-1)$.

Error bounds on the solution and a condition estimate are also provided.

In the following description, *af* stands for the subarray *af*(*iaf*:*iaf+n-1*, *jaf*:*jaf+n-1*).

The routine p?gesvx performs the following steps:

1. If *fact* = 'E', real scaling factors *R* and *C* are computed to equilibrate the system:

```
trans = 'N': diag(R) * A * diag(C) * diag(C)^-1 * X = diag(R) * B
trans = 'T': (diag(R) * A * diag(C)) T * diag(R)^-1 * X = diag(C) * B
trans = 'C': (diag(R) * A * diag(C)) H * diag(R)^-1 * X = diag(C) * B
```

Whether or not the system will be equilibrated depends on the scaling of the matrix *A*, but if equilibration is used, *A* is overwritten by *diag(R)*A*diag(C)* and *B* by *diag(R)*B* (if *trans*='N') or *diag(C)*B* (if *trans* = 'T' or 'C').

2. If *fact* = 'N' or 'E', the *LU* decomposition is used to factor the matrix *A* (after equilibration if *fact* = 'E') as *A* = *P L U*, where *P* is a permutation matrix, *L* is a unit lower triangular matrix, and *U* is upper triangular.
3. The factored form of *A* is used to estimate the condition number of the matrix *A*. If the reciprocal of the condition number is less than relative machine precision, steps 4 - 6 are skipped.
4. The system of equations is solved for *X* using the factored form of *A*.
5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
6. If equilibration was used, the matrix *X* is premultiplied by *diag(C)* (if *trans* = 'N') or *diag(R)* (if *trans* = 'T' or 'C') so that it solves the original system before equilibration.

Input Parameters

fact

(global) CHARACTER*1. Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix *A* is supplied on entry, and if not, whether the matrix *A* should be equilibrated before it is factored.

If *fact* = 'F' then, on entry, *af* and *ipiv* contain the factored form of *A*. If *equed* is not 'N', the matrix *A* has been equilibrated with scaling factors given by *r* and *c*. Arrays *a*, *af*, and *ipiv* are not modified.

If *fact* = 'N', the matrix *A* is copied to *af* and factored.

If *fact* = 'E', the matrix *A* is equilibrated if necessary, then copied to *af* and factored.

trans

(global) CHARACTER*1. Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If *trans* = 'N', the system has the form *A*X = B* (No transpose);

If *trans* = 'T', the system has the form *A^T*X = B* (Transpose);

If *trans* = 'C', the system has the form *A^H*X = B* (Conjugate transpose);

n

(global) INTEGER. The number of linear equations; the order of the submatrix *A* (*n* \geq 0).

nrhs

(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrices *B* and *X* (*nrhs* \geq 0).

a, *af*, *b*, *work*

(local)

REAL for psgesvx

DOUBLE PRECISION for pdgesvx

COMPLEX for pcgesvx

DOUBLE COMPLEX for pzgesvx.

Pointers into the local memory to arrays of local dimension $a(\text{lld_a}, \text{LOCc}(ja+n-1))$, $af(\text{lld_af}, \text{LOCc}(ja+n-1))$, $b(\text{lld_b}, \text{LOCc}(jb+nrhs-1))$, $\text{work}(\text{lwork})$, respectively.

The array a contains the matrix A . If $\text{fact} = 'F'$ and equed is not ' N ', then A must have been equilibrated by the scaling factors in r and/or c .

The array af is an input argument if $\text{fact} = 'F'$. In this case it contains on entry the factored form of the matrix A , that is, the factors L and U from the factorization $A = P^*L^*U$ as computed by [p?getrf](#). If equed is not ' N ', then af is the factored form of the equilibrated matrix A .

The array b contains on entry the matrix B whose columns are the right-hand sides for the systems of equations.

$\text{work}(*)$ is a workspace array. The dimension of work is (lwork) .

ia, ja

(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $A(ia:ia+n-1, ja:ja+n-1)$, respectively.

desca

(global and local) INTEGER array, dimension (dlen_a) . The array descriptor for the distributed matrix A .

iaf, jaf

(global) INTEGER. The row and column indices in the global array af indicating the first row and the first column of the subarray $af(iaf:iaf+n-1, jaf:jaf+n-1)$, respectively.

descaf

(global and local) INTEGER array, dimension (dlen_af) . The array descriptor for the distributed matrix AF .

ib, jb

(global) INTEGER. The row and column indices in the global array B indicating the first row and the first column of the submatrix $B(ib:ib+n-1, jb:jb+nrhs-1)$, respectively.

descb

(global and local) INTEGER array, dimension (dlen_b) . The array descriptor for the distributed matrix B .

ipiv

(local) INTEGER array.

The dimension of ipiv is $(\text{LOCr}(m_\text{a}) + mb_\text{a})$.

The array ipiv is an input argument if $\text{fact} = 'F'$.

On entry, it contains the pivot indices from the factorization $A = P^*L^*U$ as computed by [p?getrf](#); (local) row i of the matrix was interchanged with the (global) row $\text{ipiv}(i)$.

This array must be aligned with $A(ia:ia+n-1, *)$.

equed

(global) CHARACTER*1. Must be ' N ', ' R ', ' C ', or ' B '. equed is an input argument if $\text{fact} = 'F'$. It specifies the form of equilibration that was done:

If $\text{equed} = 'N'$, no equilibration was done (always true if $\text{fact} = 'N'$);

If $\text{equed} = 'R'$, row equilibration was done, that is, A has been premultiplied by $\text{diag}(r)$;
 If $\text{equed} = 'C'$, column equilibration was done, that is, A has been postmultiplied by $\text{diag}(c)$;
 If $\text{equed} = 'B'$, both row and column equilibration was done; A has been replaced by $\text{diag}(r) * A * \text{diag}(c)$.

r, c

(local) REAL for single precision flavors;
 DOUBLE PRECISION for double precision flavors.

Arrays, dimension $\text{LOC}_r(m_a)$ and $\text{LOC}_c(n_a)$, respectively.

The array r contains the row scale factors for A , and the array c contains the column scale factors for A . These arrays are input arguments if $\text{fact} = 'F'$ only; otherwise they are output arguments. If $\text{equed} = 'R'$ or $'B'$, A is multiplied on the left by $\text{diag}(r)$; if $\text{equed} = 'N'$ or $'C'$, r is not accessed.

If $\text{fact} = 'F'$ and $\text{equed} = 'R'$ or $'B'$, each element of r must be positive.

If $\text{equed} = 'C'$ or $'B'$, A is multiplied on the right by $\text{diag}(c)$; if $\text{equed} = 'N'$ or $'R'$, c is not accessed.

If $\text{fact} = 'F'$ and $\text{equed} = 'C'$ or $'B'$, each element of c must be positive. Array r is replicated in every process column, and is aligned with the distributed matrix A . Array c is replicated in every process row, and is aligned with the distributed matrix A .

ix, jx

(global) INTEGER. The row and column indices in the global array X indicating the first row and the first column of the submatrix $X(ix:ix+n-1, jx:jx+nrhs-1)$, respectively.

desc_x

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix X .

$lwork$

(local or global) INTEGER. The dimension of the array $work$; must be at least $\max(p?gecon(lwork), p?gerfs(lwork)) + \text{LOC}_r(n_a)$.

$iwork$

(local, pgesvx/pdgesvx only) INTEGER. Workspace array. The dimension of $iwork$ is ($lwork$).

$liwork$

(local, pgesvx/pdgesvx only) INTEGER. The dimension of the array $iwork$, must be at least $\text{LOC}_r(n_a)$.

$rwork$

(local) REAL for pgesvx

DOUBLE PRECISION for pzgesvx.

Workspace array, used in complex flavors only.

The dimension of $rwork$ is ($lwork$).

$lrwork$

(local or global, pgesvx/pzgesvx only) INTEGER. The dimension of the array $rwork$; must be at least $2 * \text{LOC}_c(n_a)$.

Output Parameters

x

(local)

REAL for psgesvx

DOUBLE PRECISION for pdgesvx

COMPLEX for pcgesvx

DOUBLE COMPLEX for pzgesvx.

Pointer into the local memory to an array of local dimension

 $x(\text{lld_}x, LOCC(jx+nrhs-1))$.If $\text{info} = 0$, the array x contains the solution matrix X to the *original* system of equations. Note that A and B are modified on exit if $\text{equed} \neq 'N'$, and the solution to the *equilibrated* system is: $\text{diag}(C)^{-1}X$, if $\text{trans} = 'N'$ and $\text{equed} = 'C'$ or $'B'$; and $\text{diag}(R)^{-1}X$, if $\text{trans} = 'T'$ or $'C'$ and $\text{equed} = 'R'$ or $'B'$.

a

Array a is not modified on exit if $\text{fact} = 'F'$ or $'N'$, or if $\text{fact} = 'E'$ and $\text{equed} = 'N'$.If $\text{equed} \neq 'N'$, A is scaled on exit as follows: $\text{equed} = 'R'$: $A = \text{diag}(R) * A$ $\text{equed} = 'C'$: $A = A * \text{diag}(c)$ $\text{equed} = 'B'$: $A = \text{diag}(R) * A * \text{diag}(c)$

af

If $\text{fact} = 'N'$ or $'E'$, then af is an output argument and on exit returns the factors L and U from the factorization $A = P*L*U$ of the original matrix A (if $\text{fact} = 'N'$) or of the equilibrated matrix A (if $\text{fact} = 'E'$). See the description of a for the form of the equilibrated matrix.

b

Overwritten by $\text{diag}(R)^{-1}B$ if $\text{trans} = 'N'$ and $\text{equed} = 'R'$ or $'B'$;overwritten by $\text{diag}(c)^{-1}B$ if $\text{trans} = 'T'$ and $\text{equed} = 'C'$ or $'B'$; not changed if $\text{equed} = 'N'$.

r, c

These arrays are output arguments if $\text{fact} \neq 'F'$.See the description of r, c in *Input Arguments* section.

rcond

(global) REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix A after equilibration (if done). The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

ferr, berr

(local) REAL for single precision flavors

DOUBLE PRECISION for double precision flavors.

Arrays, size $LOCC(n_b)$ each. Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

Arrays *ferr* and *berr* are both replicated in every process row, and are aligned with the matrices *B* and *X*.

ipiv

If *fact* = 'N' or 'E', then *ipiv* is an output argument and on exit contains the pivot indices from the factorization $A = P \cdot L \cdot U$ of the original matrix *A* (if *fact* = 'N') or of the equilibrated matrix *A* (if *fact* = 'E').

equed

If *fact* ≠ 'F', then *equed* is an output argument. It specifies the form of equilibration that was done (see the description of *equed* in *Input Arguments* section).

work(1)

If *info*=0, on exit *work(1)* returns the minimum value of *lwork* required for optimum performance.

iwork(1)

If *info*=0, on exit *iwork(1)* returns the minimum value of *liwork* required for optimum performance.

rwork(1)

If *info*=0, on exit *rwork(1)* returns the minimum value of *lrwork* required for optimum performance.

info

INTEGER. If *info*=0, the execution is successful.

info < 0: if the *i*th argument is an array and the *j*th entry had an illegal value, then *info* = -(*i**100+*j*); if the *i*th argument is a scalar and had an illegal value, then *info* = -*i*. If *info* = *i*, and *i* ≤ *n*, then *U(i,i)* is exactly zero. The factorization has been completed, but the factor *U* is exactly singular, so the solution and error bounds could not be computed. If *info* = *i*, and *i* = *n* + 1, then *U* is nonsingular, but *rcond* is less than machine precision. The factorization has been completed, but the matrix is singular to working precision and the solution and error bounds have not been computed.

p?gbsv

Computes the solution to the system of linear equations with a general banded distributed matrix and multiple right-hand sides.

Syntax

Fortran:

```
call psgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
call pdgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
call pcgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
call pzgbsv(n, bwl, bwu, nrhs, a, ja, desca, ipiv, b, ib, descb, work, lwork, info)
```

C:

```
void psgbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , float *a ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , float *b , MKL_INT *ib , MKL_INT
*descb , float *work , MKL_INT *lwork , MKL_INT *info );
void pdgbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , double *a ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , double *b , MKL_INT *ib , MKL_INT
*descb , double *work , MKL_INT *lwork , MKL_INT *info );
```

```
void pcgbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , MKL_Complex8
*a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );
void pzgbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , MKL_Complex16
*a , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?gbsv routine computes the solution to a real or complex system of linear equations

$\text{sub}(A) * X = \text{sub}(B)$,

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an n -by- n real/complex general banded distributed matrix with bwl subdiagonals and bwu superdiagonals, and X and $\text{sub}(B) = B(ib:ib+n-1, 1:rhs)$ are n -by- rhs distributed matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor $\text{sub}(A)$ as $\text{sub}(A) = P * L * U * Q$, where P and Q are permutation matrices, and L and U are banded lower and upper triangular matrices, respectively. The matrix Q represents reordering of columns for the sake of parallelism, while P represents reordering of rows for numerical stability using classic partial pivoting.

Input Parameters

<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>bwl</i>	(global) INTEGER. The number of subdiagonals within the band of A ($0 \leq bwl \leq n-1$).
<i>bwu</i>	(global) INTEGER. The number of superdiagonals within the band of A ($0 \leq bwu \leq n-1$).
<i>rhs</i>	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($rhs \geq 0$).
<i>a, b</i>	(local) REAL for psgbsv DOUBLE PRECISON for pdgbsv COMPLEX for pcgbsv DOUBLE COMPLEX for pzgbsv. Pointers into the local memory to arrays of local dimension $a(\text{lld}_a, \text{LOCc}(ja+n-1))$ and $b(\text{lld}_b, \text{LOCc}(rhs))$, respectively. On entry, the array <i>a</i> contains the local pieces of the global array <i>A</i> . On entry, the array <i>b</i> contains the right hand side distributed matrix $\text{sub}(B)$.
<i>ja</i>	(global) INTEGER. The index in the global array <i>A</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).

<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca(dtype_)</i> = 501, then <i>dlen_</i> ≥ 7 ; else if <i>desca(dtype_)</i> = 1, then <i>dlen_</i> ≥ 9 .
<i>ib</i>	(global) INTEGER. The row index in the global array <i>B</i> that points to the first row of the matrix to be operated on (which may be either all of <i>B</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>B</i> . If <i>descb(dtype_)</i> = 502, then <i>dlen_</i> ≥ 7 ; else if <i>descb(dtype_)</i> = 1, then <i>dlen_</i> ≥ 9 .
<i>work</i>	(local) REAL for psgbsv DOUBLE PRECISON for pdgbsv COMPLEX for pcgbsv DOUBLE COMPLEX for pzgbsv. Workspace array of dimension (<i>lwork</i>).
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i> , must be at least $\begin{aligned} lwork \geq & (NB+bwu) * (bwl+bwu) + 6 * (bwl+bwu) * (bwl+2 * bwu) + \\ & + \max(nrhs * (NB+2 * bwl+4 * bwu), 1). \end{aligned}$

Output Parameters

<i>a</i>	On exit, contains details of the factorization. Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.
<i>b</i>	On exit, this array contains the local pieces of the solution distributed matrix <i>X</i> .
<i>ipiv</i>	(local) INTEGER array. The dimension of <i>ipiv</i> must be at least <i>desca(NB)</i> . This array contains pivot indices for local factorizations. You should not alter the contents between factorization and solve.
<i>work(1)</i>	On exit, <i>work(1)</i> contains the minimum value of <i>lwork</i> required for optimum performance.
<i>info</i>	INTEGER. If <i>info</i> =0, the execution is successful. <i>info</i> < 0: If the <i>i</i> th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = $-(i*100+j)$; if the <i>i</i> th argument is a scalar and had an illegal value, then <i>info</i> = $-i$. <i>info</i> > 0:

If $info = k \leq \text{NPROCS}$, the submatrix stored on processor $info$ and factored locally was not nonsingular, and the factorization was not completed. If $info = k > \text{NPROCS}$, the submatrix stored on processor $info-\text{NPROCS}$ representing interactions with other processors was not nonsingular, and the factorization was not completed.

p?dbsv

Solves a general band system of linear equations.

Syntax

FORTRAN 77:

```
call psdbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pd dbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pc dbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pz dbsv(n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

C:

```
void psdbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , float *a ,
MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *work ,
MKL_INT *lwork , MKL_INT *info );
void pd dbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , double *a ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double
*work , MKL_INT *lwork , MKL_INT *info );
void pc dbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , MKL_Complex8
*a , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );
void pz dbsv (MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs , MKL_Complex16
*a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?dbsv routine solves the following system of linear equations:

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real/complex banded diagonally dominant-like distributed matrix with bandwidth bwl , bwu .

Gaussian elimination without pivoting is used to factor a reordering of the matrix into LU .

Input Parameters

n (global) INTEGER. The order of the distributed submatrix A , ($n \geq 0$).

bwl (global) INTEGER. Number of subdiagonals. $0 \leq bwl \leq n-1$.

<i>bwu</i>	(global) INTEGER. Number of subdiagonals. $0 \leq bwu \leq n-1$.
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B , ($nrhs \geq 0$).
<i>a</i>	(local). REAL for psdbsv DOUBLE PRECISION for pddbsv COMPLEX for pcdbsv DOUBLE COMPLEX for pzdbsv. Pointer into the local memory to an array with leading dimension $lld_a \geq (bwl+bwu+1)$ (stored in <i>desca</i>). On entry, this array contains the local pieces of the distributed matrix.
<i>ja</i>	(global) INTEGER. The index in the global array <i>a</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array of dimension <i>dlen</i> . If 1d type (<i>dtype_a</i> =501 or 502), <i>dlen</i> ≥ 7 ; If 2d type (<i>dtype_a</i> =1), <i>dlen</i> ≥ 9 . The array descriptor for the distributed matrix <i>A</i> . Contains information of mapping of <i>A</i> to memory.
<i>b</i>	(local) REAL for psdbsv DOUBLE PRECISION for pddbsv COMPLEX for pcdbsv DOUBLE COMPLEX for pzdbsv. Pointer into the local memory to an array of local lead dimension $lld_b \geq nb$. On entry, this array contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.
<i>ib</i>	(global) INTEGER. The row index in the global array <i>b</i> that points to the first row of the matrix to be operated on (which may be either all of <i>b</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array of dimension <i>dlen</i> . If 1d type (<i>dtype_b</i> =502), <i>dlen</i> ≥ 7 ; If 2d type (<i>dtype_b</i> =1), <i>dlen</i> ≥ 9 . The array descriptor for the distributed matrix <i>B</i> . Contains information of mapping of <i>B</i> to memory.
<i>work</i>	(local). REAL for psdbsv DOUBLE PRECISION for pddbsv

COMPLEX for `pcdbsv`

DOUBLE COMPLEX for `pzdbsv`.

Temporary workspace. This space may be overwritten in between calls to routines. `work` must be the size given in `lwork`.

`lwork`

(local or global) INTEGER. Size of user-input workspace `work`. If `lwork` is too small, the minimal acceptable size will be returned in `work(1)` and an error code is returned.

$$\begin{aligned} lwork \geq nb(bwl+bwu) + & 6\max(bwl, bwu) * \max(bwl, bwu) \\ & + \max(\max(bwl, bwu) nrhs), \quad \max(bwl, bwu) * \max(bwl, bwu) \end{aligned}$$

Output Parameters

`a`

On exit, this array contains information containing details of the factorization.

Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

`b`

On exit, this contains the local piece of the solutions distributed matrix X .

`work`

On exit, `work(1)` contains the minimal `lwork`.

`info`

(local) INTEGER. If `info`=0, the execution is successful.

< 0: If the i -th argument is an array and the j -entry had an illegal value, then `info` = $-(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then `info` = $-i$.

> 0: If `info` = k < NPROCS, the submatrix stored on processor `info` and factored locally was not positive definite, and the factorization was not completed.

If `info` = k > NPROCS, the submatrix stored on processor `info`-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?dtsv

Solves a general tridiagonal system of linear equations.

Syntax

Fortran:

```
call psdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pddtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pcdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
call pzdtsv(n, nrhs, dl, d, du, ja, desca, b, ib, descb, work, lwork, info)
```

C:

```
void psdtsv (MKL_INT *n , MKL_INT *nrhs , float *dl , float *d , float *du , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *work , MKL_INT
*iwork , MKL_INT *info );
```

```

void pddtsv (MKL_INT *n , MKL_INT *nrhs , double *dl , double *d , double *du ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double
*work , MKL_INT *lwork , MKL_INT *info );

void pcdtsv (MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *dl , MKL_Complex8 *d ,
MKL_Complex8 *du , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzdtsv (MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *dl , MKL_Complex16 *d ,
MKL_Complex16 *du , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The routine solves a system of linear equations

$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs)$,

where $A(1:n, ja:ja+n-1)$ is an n -by- n complex tridiagonal diagonally dominant-like distributed matrix.

Gaussian elimination without pivoting is used to factor a reordering of the matrix into $L U$.

Input Parameters

<i>n</i>	(global) INTEGER. The order of the distributed submatrix A ($n \geq 0$).
<i>nrhs</i>	INTEGER. The number of right hand sides; the number of columns of the distributed matrix B ($nrhs \geq 0$).
<i>dl</i>	(local). REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer to local part of global vector storing the lower diagonal of the matrix. Globally, $d/(1)$ is not referenced, and dl must be aligned with d . Must be of size $> desca(nb_)$.
<i>d</i>	(local). REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer to local part of global vector storing the main diagonal of the matrix.
<i>du</i>	(local). REAL for psdtsv DOUBLE PRECISION for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, $du(n)$ is not referenced, and du must be aligned with d .

<i>ja</i>	(global) INTEGER. The index in the global array <i>a</i> that points to the start of the matrix to be operated on (which may be either all of <i>A</i> or a submatrix of <i>A</i>).
<i>desca</i>	(global and local) INTEGER array of dimension <i>dlen</i> . If 1d type (<i>dtype_a</i> =501 or 502), <i>dlen</i> ≥ 7 ; If 2d type (<i>dtype_a</i> =1), <i>dlen</i> ≥ 9 . The array descriptor for the distributed matrix <i>A</i> . Contains information of mapping of <i>A</i> to memory.
<i>b</i>	(local) REAL for psdtsv DOUBLE PRECISON for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Pointer into the local memory to an array of local lead dimension <i>lld_b</i> > <i>nb</i> . On entry, this array contains the local pieces of the right hand sides <i>B</i> (<i>ib</i> : <i>ib+n-1</i> , 1: <i>nrhs</i>).
<i>ib</i>	(global) INTEGER. The row index in the global array <i>b</i> that points to the first row of the matrix to be operated on (which may be either all of <i>b</i> or a submatrix of <i>B</i>).
<i>descb</i>	(global and local) INTEGER array of dimension <i>dlen</i> . If 1d type (<i>dtype_b</i> =502), <i>dlen</i> ≥ 7 ; If 2d type (<i>dtype_b</i> =1), <i>dlen</i> ≥ 9 . The array descriptor for the distributed matrix <i>B</i> . Contains information of mapping of <i>B</i> to memory.
<i>work</i>	(local). REAL for psdtsv DOUBLE PRECISON for pddtsv COMPLEX for pcdtsv DOUBLE COMPLEX for pzdtsv. Temporary workspace. This space may be overwritten in between calls to routines. <i>work</i> must be the size given in <i>lwork</i> .
<i>lwork</i>	(local or global) INTEGER. Size of user-input workspace <i>work</i> . If <i>lwork</i> is too small, the minimal acceptable size will be returned in <i>work</i> (1) and an error code is returned. <i>lwork</i> $>$ (12*NPCOL+3* <i>nb</i>) +max((10+2*min(100, <i>nrhs</i>))*NPCOL+4* <i>nrhs</i> , 8*NPCOL)

Output Parameters

<i>dl</i>	On exit, this array contains information containing the * factors of the matrix.
-----------	--

<i>d</i>	On exit, this array contains information containing the * factors of the matrix. Must be of size > <i>desca(nb_)</i> .
<i>du</i>	On exit, this array contains information containing the * factors of the matrix. Must be of size > <i>desca(nb_)</i> .
<i>b</i>	On exit, this contains the local piece of the solutions distributed matrix <i>X</i> .
<i>work</i>	On exit, <i>work(1)</i> contains the minimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. If <i>info</i> =0, the execution is successful. < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . > 0: If <i>info</i> = <i>k</i> < NPROCS, the submatrix stored on processor <i>info</i> and factored locally was not positive definite, and the factorization was not completed. If <i>info</i> = <i>k</i> > NPROCS, the submatrix stored on processor <i>info</i> -NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?posv

Solves a symmetric positive definite system of linear equations.

Syntax

Fortran:

```
call psposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pdposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pcposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
call pzposv(uplo, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, info)
```

C:

```
void psposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
MKL_INT *info );
void pdposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
void pcposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
void pzposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?posv` routine computes the solution to a real/complex system of linear equations

$$\text{sub}(A) * X = \text{sub}(B),$$

where `sub(A)` denotes $A(ia:ia+n-1, ja:ja+n-1)$ and is an n -by- n symmetric/Hermitian distributed positive definite matrix and X and `sub(B)` denoting $B(ib:ib+n-1, jb:jb+nrhs-1)$ are n -by- $nrhs$ distributed matrices. The Cholesky decomposition is used to factor `sub(A)` as

$$\text{sub}(A) = U^T * U, \text{ if } \text{uplo} = 'U', \text{ or}$$

$$\text{sub}(A) = L * L^T, \text{ if } \text{uplo} = 'L',$$

where U is an upper triangular matrix and L is a lower triangular matrix. The factored form of `sub(A)` is then used to solve the system of equations.

Input Parameters

<code>uplo</code>	(global) CHARACTER. Must be ' <code>U</code> ' or ' <code>L</code> '.
	Indicates whether the upper or lower triangular part of <code>sub(A)</code> is stored.
<code>n</code>	(global) INTEGER. The order of the distributed submatrix <code>sub(A)</code> ($n \geq 0$).
<code>nrhs</code>	INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix <code>sub(B)</code> ($nrhs \geq 0$).
<code>a</code>	(local) REAL for <code>psposv</code> DOUBLE PRECISION for <code>pdposv</code> COMPLEX for <code>pcposv</code> COMPLEX*16 for <code>pzposv</code> . Pointer into the local memory to an array of dimension (<code>lld_a</code> , <code>LOCc(ja+n-1)</code>). On entry, this array contains the local pieces of the n -by- n symmetric distributed matrix <code>sub(A)</code> to be factored. If <code>uplo</code> = ' <code>U</code> ', the leading n -by- n upper triangular part of <code>sub(A)</code> contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If <code>uplo</code> = ' <code>L</code> ', the leading n -by- n lower triangular part of <code>sub(A)</code> contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array <code>a</code> indicating the first row and the first column of the submatrix <code>A</code> , respectively.
<code>desca</code>	(global and local) INTEGER array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix <code>A</code> .
<code>b</code>	(local) REAL for <code>psposv</code>

DOUBLE PRECISION for pdposv

COMPLEX for pcposv

COMPLEX*16 for pzposv.

Pointer into the local memory to an array of dimension ($11d_b, LOC(jb) + nrhs - 1$). On entry, the local pieces of the right hand sides distributed matrix sub(B).

ib, jb

(global) INTEGER. The row and column indices in the global array b indicating the first row and the first column of the submatrix B , respectively.

$descb$

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix B .

Output Parameters

a

On exit, if $info = 0$, this array contains the local pieces of the factor U or L from the Cholesky factorization $\text{sub}(A) = U^H * U$, or $L * L^H$.

b

On exit, if $info = 0$, $\text{sub}(B)$ is overwritten by the solution distributed matrix X .

$info$

(global) INTEGER.

If $info = 0$, the execution is successful.

If $info < 0$: If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

If $info > 0$: If $info = k$, the leading minor of order k , $A(ia:ia+k-1, ja:ja+k-1)$ is not positive definite, and the factorization could not be completed, and the solution has not been computed.

p?posvx

Solves a symmetric or Hermitian positive definite system of linear equations.

Syntax

Fortran:

```
call pspovsx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, equeed, sr,
sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork, iwork,
liwork, info)

call pdposvx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, equeed, sr,
sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork, iwork,
liwork, info)

call pcposvx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, equeed, sr,
sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork, iwork,
liwork, info)
```

```
call pzposvx(fact, uplo, n, nrhs, a, ia, ja, desca, af, iaf, jaf, descaf, equed, sr,
sc, b, ib, jb, descb, x, ix, jx, descx, rcond, ferr, berr, work, lwork, iwork,
liwork, info)
```

C:

```
void pspovsx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *iaf , MKL_INT *jaf ,
MKL_INT *descf , char *equed , float *sr , float *sc , float *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
float *rcond , float *ferr , float *berr , float *work , MKL_INT *lwork , MKL_INT
*iwork , MKL_INT *liwork , MKL_INT *info );

void pdposvx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *af , MKL_INT *iaf , MKL_INT
*jaf , MKL_INT *descf , char *equed , double *sr , double *sc , double *b , MKL_INT
*ib , MKL_INT *jb , MKL_INT *descb , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , double *rcond , double *ferr , double *berr , double *work , MKL_INT *lwork ,
MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pcposvx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *iaf , MKL_INT
*jaf , MKL_INT *descf , char *equed , float *sr , float *sc , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , float *rcond , float *ferr , float *berr , MKL_Complex8 *work ,
MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *info );

void pzposvx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *iaf ,
MKL_INT *jaf , MKL_INT *descf , char *equed , double *sr , double *sc , MKL_Complex16
*b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , double *rcond , double *ferr , double *berr ,
MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT *lrwork , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?posvx` routine uses the Cholesky factorization $A=U^T \cdot U$ or $A=L \cdot L^T$ to compute the solution to a real or complex system of linear equations

$$A(ia:ia+n-1, ja:ja+n-1) \cdot X = B(ib:ib+n-1, jb:jb+nrhs-1),$$

where $A(ia:ia+n-1, ja:ja+n-1)$ is a n -by- n matrix and X and $B(ib:ib+n-1, jb:jb+nrhs-1)$ are n -by- $nrhs$ matrices.

Error bounds on the solution and a condition estimate are also provided.

In the following comments y denotes $Y(iy:iy+m-1, jy:jy+k-1)$ a m -by- k matrix where y can be a , af , b and x .

The routine `p?posvx` performs the following steps:

1. If $fact = 'E'$, real scaling factors s are computed to equilibrate the system:

$$\text{diag}(sr) \cdot A \cdot \text{diag}(sc) \cdot \text{inv}(\text{diag}(sc)) \cdot X = \text{diag}(sr) \cdot B$$

Whether or not the system will be equilibrated depends on the scaling of the matrix A , but if equilibration is used, A is overwritten by $\text{diag}(sr) \cdot A \cdot \text{diag}(sc)$ and B by $\text{diag}(sr) \cdot B$.

- 2.** If $\text{fact} = \text{'N'}$ or 'E' , the Cholesky decomposition is used to factor the matrix A (after equilibration if $\text{fact} = \text{'E'}$) as
- $$A = U^T * U, \text{ if } \text{uplo} = \text{'U'}, \text{ or}$$
- $$A = L * L^T, \text{ if } \text{uplo} = \text{'L'},$$
- where U is an upper triangular matrix and L is a lower triangular matrix.
- 3.** The factored form of A is used to estimate the condition number of the matrix A . If the reciprocal of the condition number is less than machine precision, steps 4-6 are skipped
- 4.** The system of equations is solved for X using the factored form of A .
- 5.** Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
- 6.** If equilibration was used, the matrix X is premultiplied by $\text{diag}(sr)$ so that it solves the original system before equilibration.

Input Parameters

<i>fact</i>	(global) CHARACTER. Must be ' <i>F</i> ', ' <i>N</i> ', or ' <i>E</i> '.
	Specifies whether or not the factored form of the matrix A is supplied on entry, and if not, whether the matrix A should be equilibrated before it is factored.
	If $\text{fact} = \text{'F'}$: on entry, <i>af</i> contains the factored form of A . If <i>equed</i> = ' <i>Y</i> ', the matrix A has been equilibrated with scaling factors given by <i>s</i> . <i>a</i> and <i>af</i> will not be modified.
	If $\text{fact} = \text{'N'}$, the matrix A will be copied to <i>af</i> and factored.
	If $\text{fact} = \text{'E'}$, the matrix A will be equilibrated if necessary, then copied to <i>af</i> and factored.
<i>uplo</i>	(global) CHARACTER. Must be ' <i>U</i> ' or ' <i>L</i> '.
	Indicates whether the upper or lower triangular part of A is stored.
<i>n</i>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$ ($n \geq 0$).
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrices B and X . ($nrhs \geq 0$).
<i>a</i>	(local) REAL for <i>psposvx</i> DOUBLE PRECISION for <i>pdpovx</i> COMPLEX for <i>pcposvx</i> DOUBLE COMPLEX for <i>pzposvx</i> . Pointer into the local memory to an array of local dimension (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). On entry, the symmetric/Hermitian matrix A , except if $\text{fact} = \text{'F'}$ and <i>equed</i> = ' <i>Y</i> ', then A must contain the equilibrated matrix $\text{diag}(sr) * A * \text{diag}(sc)$. If <i>uplo</i> = ' <i>U</i> ', the leading n -by- n upper triangular part of A contains the upper triangular part of the matrix A , and the strictly lower triangular part of A is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of A contains the lower triangular part of the matrix A , and the strictly upper triangular part of A is not referenced. A is not modified if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$ on exit.

ia, ja

(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

desca

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A .

af

(local)

REAL for `psposvx`

DOUBLE PRECISION for `pdpovsx`

COMPLEX for `pcposvx`

DOUBLE COMPLEX for `pzposvx`.

Pointer into the local memory to an array of local dimension (lld_af , $LOCc(ja+n-1)$).

If $fact = 'F'$, then af is an input argument and on entry contains the triangular factor U or L from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$, in the same storage format as A . If $equed \neq 'N'$, then af is the factored form of the equilibrated matrix $\text{diag}(sr) * A * \text{diag}(sc)$.

iaf, jaf

(global) INTEGER. The row and column indices in the global array af indicating the first row and the first column of the submatrix AF , respectively.

descaf

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix AF .

equed

(global) CHARACTER. Must be $'N'$ or $'Y'$.

equed is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:

If $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$);

If $equed = 'Y'$, equilibration was done and A has been replaced by $\text{diag}(sr) * A * \text{diag}(sc)$.

sr

(local)

REAL for `psposvx`

DOUBLE PRECISION for `pdpovsx`

COMPLEX for `pcposvx`

DOUBLE COMPLEX for `pzposvx`.

Array, size (lld_a).

The array s contains the scale factors for A . This array is an input argument if $fact = 'F'$ only; otherwise it is an output argument.

If $equed = 'N'$, s is not accessed.

If `fact = 'F'` and `equed = 'Y'`, each element of `s` must be positive.

b

(local)

REAL for `psposvx`DOUBLE PRECISION for `pdpovsx`COMPLEX for `pcposvx`DOUBLE COMPLEX for `pzposvx`.

Pointer into the local memory to an array of local dimension (`lld_b`, `LOCc(jb+nrhs-1)`). On entry, the *n*-by-*rhs* right-hand side matrix *B*.

ib, jb

(global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb

(global and local) INTEGER. Array, dimension (`dlen_`). The array descriptor for the distributed matrix *B*.

x

(local)

REAL for `psposvx`DOUBLE PRECISION for `pdpovsx`COMPLEX for `pcposvx`DOUBLE COMPLEX for `pzposvx`.

Pointer into the local memory to an array of local dimension (`lld_x`, `LOCc(jx+nrhs-1)`).

ix, jx

(global) INTEGER. The row and column indices in the global array *x* indicating the first row and the first column of the submatrix *X*, respectively.

descx

(global and local) INTEGER array, dimension (`dlen_`). The array descriptor for the distributed matrix *X*.

work

(local)

REAL for `psposvx`DOUBLE PRECISION for `pdpovsx`COMPLEX for `pcposvx`DOUBLE COMPLEX for `pzposvx`.Workspace array, size (`lwork`).*lwork*

(local or global) INTEGER.

The dimension of the array *work*. *lwork* is local input and must be at least $\text{lwork} = \max(\text{p?pocon}(\text{lwork}), \text{p?porfs}(\text{lwork})) + \text{LOCr}(n_a)$.

*lwork = 3*desca(lld_)*.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

 $iwork$ (local) INTEGER. Workspace array, dimension ($lwork$). $lwork$

(local or global)

INTEGER. The dimension of the array $iwork$. $lwork$ is local input and must be at least $lwork = \text{desc}(1ld_1) lwork = \text{LOCr}(n_a)$.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

 a

On exit, if $\text{fact} = 'E'$ and $\text{equed} = 'Y'$, a is overwritten by $\text{diag}(sr) * a * \text{diag}(sc)$.

 af

If $\text{fact} = 'N'$, then af is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ of the original matrix A .

If $\text{fact} = 'E'$, then af is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A = U^T * U$ or $A = L * L^T$ of the equilibrated matrix A (see the description of A for the form of the equilibrated matrix).

 $equed$

If $\text{fact} \neq 'F'$, then $equed$ is an output argument. It specifies the form of equilibration that was done (see the description of $equed$ in *Input Arguments* section).

 sr

This array is an output argument if $\text{fact} \neq 'F'$.

See the description of sr in *Input Arguments* section.

 sc

This array is an output argument if $\text{fact} \neq 'F'$.

See the description of sc in *Input Arguments* section.

 b

On exit, if $equed = 'N'$, b is not modified; if $\text{trans} = 'N'$ and $equed = 'R'$ or $'B'$, b is overwritten by $\text{diag}(r) * b$; if $\text{trans} = 'T'$ or $'C'$ and $equed = 'C'$ or $'B'$, b is overwritten by $\text{diag}(c) * b$.

 x

(local)

REAL for `psposvx`DOUBLE PRECISION for `pdposvx`COMPLEX for `pcposvx`DOUBLE COMPLEX for `pzposvx`.

If $info = 0$ the n -by- $nrhs$ solution matrix X to the original system of equations.

Note that A and B are modified on exit if $\text{equed} \neq 'N'$, and the solution to the equilibrated system is

```
inv(diag( $sc$ )) $*X$  if  $\text{trans} = 'N'$  and  $\text{equed} = 'C'$  or ' $B$ ', or  
inv(diag( $sr$ )) $*X$  if  $\text{trans} = 'T'$  or ' $C$ ' and  $\text{equed} = 'R'$  or ' $B$ '.
```

rcond

(global)

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

An estimate of the reciprocal condition number of the matrix A after equilibration (if done). If $rcond$ is less than the machine precision (in particular, if $rcond=0$), the matrix is singular to working precision. This condition is indicated by a return code of $\text{info} > 0$.

ferr

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays, size at least $\max(LOC, n_b)$. The estimated forward error bounds for each solution vector $X(j)$ (the j -th column of the solution matrix X). If $xtrue$ is the true solution, $ferr(j)$ bounds the magnitude of the largest entry in $(X(j) - xtrue)$ divided by the magnitude of the largest entry in $X(j)$. The quality of the error bound depends on the quality of the estimate of $\text{norm}(\text{inv}(A))$ computed in the code; if the estimate of $\text{norm}(\text{inv}(A))$ is accurate, the error bound is guaranteed.

berr

(local)

REAL for single precision flavors.

DOUBLE PRECISION for double precision flavors.

Arrays, size at least $\max(LOC, n_b)$. The componentwise relative backward error of each solution vector $X(j)$ (the smallest relative change in any entry of A or B that makes $X(j)$ an exact solution).

work(1)(local) On exit, *work(1)* returns the minimal and optimal *lwork*.*info*

(global) INTEGER.

If $\text{info}=0$, the execution is successful.< 0: if $\text{info} = -i$, the i -th argument had an illegal value> 0: if $\text{info} = i$, and $i \leq n$: if $\text{info} = i$, the leading minor of order i of A is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed.= $n+1$: $rcond$ is less than machine precision. The factorization has been completed, but the matrix is singular to working precision, and the solution and error bounds have not been computed.

p?pbsv

Solves a symmetric/Hermitian positive definite banded system of linear equations.

Syntax

Fortran:

```
call pspbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pdpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pcpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
call pzpbsv(uplo, n, bw, nrhs, a, ja, desca, b, ib, descb, work, lwork, info)
```

C:

```
void pspbsv (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , float *a ,
MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *work ,
MKL_INT *lwork , MKL_INT *info );
void pdpbsv (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , double *a ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double
*work , MKL_INT *lwork , MKL_INT *info );
void pcpbsv (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , MKL_Complex8 *a ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );
void pzpbsv (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , MKL_Complex16 *a ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?pbsv routine solves a system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real/complex banded symmetric positive definite distributed matrix with bandwidth bw .

Cholesky factorization is used to factor a reordering of the matrix into L^*L' .

Input Parameters

<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'.
	Indicates whether the upper or lower triangular of A is stored.
	If <i>uplo</i> = 'U', the upper triangular A is stored
	If <i>uplo</i> = 'L', the lower triangular of A is stored.
<i>n</i>	(global) INTEGER. The order of the distributed matrix A ($n \geq 0$).
<i>bw</i>	(global) INTEGER. The number of subdiagonals in L or U . $0 \leq bw \leq n-1$.
<i>nrhs</i>	(global) INTEGER. The number of right-hand sides; the number of columns in B ($nrhs \geq 0$).
<i>a</i>	(local). REAL for pspbsv DOUBLE PRECISON for pdpbsv

COMPLEX for `pcpbsv`

DOUBLE COMPLEX for `pzpbsv`.

Pointer into the local memory to an array with leading dimension $lld_a \geq (bw+1)$ (stored in `desca`). On entry, this array contains the local pieces of the distributed matrix `sub(A)` to be factored.

ja
 (global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca
 (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*.

b
 (local)

REAL for `pspbsv`

DOUBLE PRECISON for `pdpbsv`

COMPLEX for `pcpbsv`

DOUBLE COMPLEX for `pzpbsv`.

Pointer into the local memory to an array of local lead dimension $lld_b \geq nb$. On entry, this array contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.

ib
 (global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

descb
 (global and local) INTEGER array of dimension *dlen*.

If 1D type (`dtype_b =502`), *dlen* ≥ 7 ;

If 2D type (`dtype_b =1`), *dlen* ≥ 9 .

The array descriptor for the distributed matrix *B*.

Contains information of mapping of *B* to memory.

work
 (local).

REAL for `pspbsv`

DOUBLE PRECISON for `pdpbsv`

COMPLEX for `pcpbsv`

DOUBLE COMPLEX for `pzpbsv`.

Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork
 (local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work(1)* and an error code is returned. $lwork \geq (nb+2*bw)*bw + \max((bw*nrhs), bw*bw)$

Output Parameters

<i>a</i>	On exit, this array contains information containing details of the factorization. Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.
<i>b</i>	On exit, contains the local piece of the solutions distributed matrix <i>X</i> .
<i>work</i>	On exit, <i>work</i> (1) contains the minimal <i>lwork</i> .
<i>info</i>	(global) INTEGER. If <i>info</i> =0, the execution is successful. < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . > 0: If <i>info</i> = <i>k</i> ≤ NPROCS, the submatrix stored on processor <i>info</i> and factored locally was not positive definite, and the factorization was not completed. If <i>info</i> = <i>k</i> > NPROCS, the submatrix stored on processor <i>info</i> -NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

p?ptsv

Syntax

Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations.

Fortran:

```
call psptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pdptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pcptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
call pzptsv(n, nrhs, d, e, ja, desca, b, ib, descb, work, lwork, info)
```

C:

```
void psptsv (MKL_INT *n , MKL_INT *nrhs , float *d , float *e , MKL_INT *ja , MKL_INT
*desca , float *b , MKL_INT *ib , MKL_INT *descb , float *work , MKL_INT *lwork ,
MKL_INT *info );
void pdptsv (MKL_INT *n , MKL_INT *nrhs , double *d , double *e , MKL_INT *ja ,
MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double *work , MKL_INT
*lwork , MKL_INT *info );
void pcptsv (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *d , MKL_Complex8 *e ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );
void pzptsv (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *d , MKL_Complex16 *e ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?ptsv` routine solves a system of linear equations

$$A(1:n, ja:ja+n-1)*X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an n -by- n real tridiagonal symmetric positive definite distributed matrix.

Cholesky factorization is used to factor a reordering of the matrix into $L*L'$.

Input Parameters

<code>n</code>	(global) INTEGER. The order of matrix A ($n \geq 0$).
<code>nrhs</code>	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B ($nrhs \geq 0$).
<code>d</code>	(local) REAL for <code>psptsv</code> DOUBLE PRECISON for <code>pdptsv</code> COMPLEX for <code>pcptsv</code> DOUBLE COMPLEX for <code>pzptsv</code> . Pointer to local part of global vector storing the main diagonal of the matrix.
<code>e</code>	(local) REAL for <code>psptsv</code> DOUBLE PRECISON for <code>pdptsv</code> COMPLEX for <code>pcptsv</code> DOUBLE COMPLEX for <code>pzptsv</code> . Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, $du(n)$ is not referenced, and du must be aligned with d .
<code>ja</code>	(global) INTEGER. The index in the global array A that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<code>desca</code>	(global and local) INTEGER array of dimension <code>dlen</code> . If 1d type (<code>dtype_a=501 or 502</code>), <code>dlen</code> ≥ 7 ; If 2d type (<code>dtype_a=1</code>), <code>dlen</code> ≥ 9 . The array descriptor for the distributed matrix A . Contains information of mapping of A to memory.
<code>b</code>	(local) REAL for <code>psptsv</code> DOUBLE PRECISON for <code>pdptsv</code> COMPLEX for <code>pcptsv</code> DOUBLE COMPLEX for <code>pzptsv</code> .

Pointer into the local memory to an array of local lead dimension $l1d_b \geq nb$.

On entry, this array contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.

ib (global) INTEGER. The row index in the global array b that points to the first row of the matrix to be operated on (which may be either all of b or a submatrix of B).

descb (global and local) INTEGER array of dimension *dlen*.

If 1d type ($dtype_b = 502$), *dlen* ≥ 7 ;

If 2d type ($dtype_b = 1$), *dlen* ≥ 9 .

The array descriptor for the distributed matrix B .

Contains information of mapping of B to memory.

work (local).

REAL for psptsv

DOUBLE PRECISON for pdptsv

COMPLEX for pcptsv

DOUBLE COMPLEX for pzptsv.

Temporary workspace. This space may be overwritten in between calls to routines. *work* must be the size given in *lwork*.

lwork (local or global) INTEGER. Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned. $lwork > (12 * \text{NPCOL} + 3 * nb) + \max((10 + 2 * \min(100, nrhs)) * \text{NPCOL} + 4 * nrhs, 8 * \text{NPCOL})$.

Output Parameters

d On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to *desca*(*nb*_).

e On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to *desca*(*nb*_).

b On exit, this contains the local piece of the solutions distributed matrix X .

work On exit, *work*(1) contains the minimal *lwork*.

info (local) INTEGER. If *info*=0, the execution is successful.

< 0: If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i * 100 + j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

> 0: If $info = k \leq \text{NPROCS}$, the submatrix stored on processor *info* and factored locally was not positive definite, and the factorization was not completed.

If $info = k > \text{NPROCS}$, the submatrix stored on processor $info\text{-NPROCS}$ representing interactions with other processors was not positive definite, and the factorization was not completed.

p?gels

Solves overdetermined or underdetermined linear systems involving a matrix of full rank.

Syntax

Fortran:

```
call psgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
call pdgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
call pcgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
call pzgels(trans, m, n, nrhs, a, ia, ja, desca, b, ib, jb, descb, work, lwork, info)
```

C:

```
void psgels (char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs , float *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , float *work , MKL_INT *lwork , MKL_INT *info );
void pdgels (char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs , double *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , double *work , MKL_INT *lwork , MKL_INT *info );
void pcgels (char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT
*jb , MKL_INT *descb , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );
void pzgels (char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT
*jb , MKL_INT *descb , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?gels routine solves overdetermined or underdetermined real/ complex linear systems involving an m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$, or its transpose/ conjugate-transpose, using a QTQ or LQ factorization of $\text{sub}(A)$. It is assumed that $\text{sub}(A)$ has full rank.

The following options are provided:

1. If $trans = 'N'$ and $m \geq n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem
$$\text{minimize } ||\text{sub}(B) - \text{sub}(A) * X||$$
2. If $trans = 'N'$ and $m < n$: find the minimum norm solution of an underdetermined system $\text{sub}(A) * X = \text{sub}(B)$.
3. If $trans = 'T'$ and $m \geq n$: find the minimum norm solution of an undetermined system $\text{sub}(A)^T * X = \text{sub}(B)$.
4. If $trans = 'T'$ and $m < n$: find the least squares solution of an overdetermined system, that is, solve the least squares problem

```
minimize ||sub(B) - sub(A)T*X||,
```

where `sub(B)` denotes $B(ib:ib+m-1, jb:jb+nrhs-1)$ when `trans = 'N'` and $B(ib:ib+n-1, jb:jb+nrhs-1)$ otherwise. Several right hand side vectors b and solution vectors x can be handled in a single call; when `trans = 'N'`, the solution vectors are stored as the columns of the n -by- $nrhs$ right hand side matrix `sub(B)` and the m -by- $nrhs$ right hand side matrix `sub(B)` otherwise.

Input Parameters

<code>trans</code>	(global) CHARACTER. Must be ' <code>N</code> ', or ' <code>T</code> '. If <code>trans = 'N'</code> , the linear system involves matrix <code>sub(A)</code> ; If <code>trans = 'T'</code> , the linear system involves the transposed matrix A^T (for real flavors only).
<code>m</code>	(global) INTEGER. The number of rows in the distributed submatrix <code>sub(A)</code> ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns in the distributed submatrix <code>sub(A)</code> ($n \geq 0$).
<code>nrhs</code>	(global) INTEGER. The number of right-hand sides; the number of columns in the distributed submatrices <code>sub(B)</code> and <code>X</code> . ($nrhs \geq 0$).
<code>a</code>	(local) REAL for <code>psgels</code> DOUBLE PRECISION for <code>pdgels</code> COMPLEX for <code>pcgels</code> DOUBLE COMPLEX for <code>pzgels</code> . Pointer into the local memory to an array of dimension (<code>lld_a, LOCc(ja+n-1)</code>). On entry, contains the m -by- n matrix A .
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array <code>a</code> indicating the first row and the first column of the submatrix A , respectively.
<code>desca</code>	(global and local) INTEGER array, dimension (<code>dlen_</code>). The array descriptor for the distributed matrix A .
<code>b</code>	(local) REAL for <code>psgels</code> DOUBLE PRECISION for <code>pdgels</code> COMPLEX for <code>pcgels</code> DOUBLE COMPLEX for <code>pzgels</code> . Pointer into the local memory to an array of local dimension (<code>lld_b, LOCc(jb+nrhs-1)</code>). On entry, this array contains the local pieces of the distributed matrix B of right-hand side vectors, stored columnwise; <code>sub(B)</code> is m -by- $nrhs$ if <code>trans='N'</code> , and n -by- $nrhs$ otherwise.

ib, jb (global) INTEGER. The row and column indices in the global array *b* indicating the first row and the first column of the submatrix *B*, respectively.

descb (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *B*.

work (local)

REAL for psgels
DOUBLE PRECISION for pdgels
COMPLEX for pcgels
DOUBLE COMPLEX for pzgels.

Workspace array with dimension *lwork*.

lwork (local or global) INTEGER.

The dimension of the array *work lwork* is local input and must be at least $lwork \geq ltau + \max(lwf, lws)$, where if $m > n$, then

```

ltau = numroc(ja+min(m,n)-1, nb_a, MYCOL, csrc_a, NPCOL),
lwf = nb_a * (mpa0 + nqa0 + nb_a)
lws = max((nb_a * (nb_a-1))/2, (nrhsqb0 + mpb0) * nb_a) +
      nb_a * nb_a
else
    ltau = numroc(ia+min(m,n)-1, mb_a, MYROW, rsrc_a, NPROW),
    lwf = mb_a * (mpa0 + nqa0 + mb_a)
    lws = max((mb_a * (mb_a-1))/2, (npb0 + max(nqa0 +
        numroc(numroc(n+iroffb, mb_a, 0, 0, NPROW), mb_a, 0, 0,
        lcm), nrhsqb0)) * mb_a) + mb_a * mb_a
end if,
where lcm = lcm/NPROW with lcm = ilcm(NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icooffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
iacol= indxg2p(ja, nb_a, MYROW, rsrc_a, NPROW)
mpa0 = numroc(m+iroffa, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+icooffa, nb_a, MYCOL, iacol, NPCOL),
iroffb = mod(ib-1, mb_b),
icoffb = mod(jb-1, nb_b),
ibrow = indxg2p(ib, mb_b, MYROW, rsrc_b, NPROW),
ibcol = indxg2p(jb, nb_b, MYCOL, csrc_b, NPCOL),
mpb0 = numroc(m+iroffb, mb_b, MYROW, icrow, NPROW),
nqb0 = numroc(n+icoffb, nb_b, MYCOL, ibcol, NPCOL),

```

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW`, and `NPCOL` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

On exit, If $m \geq n$, `sub(A)` is overwritten by the details of its *QR* factorization as returned by `p?geqrf`; if $m < n$, `sub(A)` is overwritten by details of its *LQ* factorization as returned by `p?gelqf`.

b

On exit, `sub(B)` is overwritten by the solution vectors, stored columnwise: if `trans = 'N'` and $m \geq n$, rows 1 to n of `sub(B)` contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements $n+1$ to m in that column;

If `trans = 'N'` and $m < n$, rows 1 to n of `sub(B)` contain the minimum norm solution vectors;

If `trans = 'T'` and $m \geq n$, rows 1 to m of `sub(B)` contain the minimum norm solution vectors; if `trans = 'T'` and $m < n$, rows 1 to m of `sub(B)` contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements $m+1$ to n in that column.

`work(1)`

On exit, `work(1)` contains the minimum value of `lwork` required for optimum performance.

`info`

(global) INTEGER.

= 0: the execution is successful.

< 0: if the i -th argument is an array and the j -entry had an illegal value, then `info = - (i * 100 + j)`, if the i -th argument is a scalar and had an illegal value, then `info = -i`.

p?syev

Computes selected eigenvalues and eigenvectors of a symmetric matrix.

Syntax

Fortran:

```
call pssyev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, info)
call pdsyev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, info)
```

C:

```
void pssyev (char *jobz , char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *w , float *z , MKL_INT *iz , MKL_INT *jz , MKL_INT
*descz , float *work , MKL_INT *lwork , MKL_INT *info );
```

```
void pdssyev (char *jobz , char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *w , double *z , MKL_INT *iz , MKL_INT *jz , MKL_INT
*descz , double *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?syev routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A by calling the recommended sequence of ScaLAPACK routines.

In its present form, the routine assumes a homogeneous system and makes no checks for consistency of the eigenvalues or eigenvectors across the different processes. Because of this, it is possible that a heterogeneous system may return incorrect results without any error messages.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global) CHARACTER. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors:

If *jobz* ='N', then only eigenvalues are computed.

If *jobz* ='V', then eigenvalues and eigenvectors are computed.

uplo (global) CHARACTER. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

If *uplo* = 'U', *a* stores the upper triangular part of A .

If *uplo* = 'L', *a* stores the lower triangular part of A .

n (global) INTEGER. The number of rows and columns of the matrix A ($n \geq 0$).

a (local)

REAL for pdssyev.

DOUBLE PRECISION for pdssyev.

Block cyclic array of global dimension (n, n) and local dimension (*lld_a*, LOC $c(ja+n-1)$). On entry, the symmetric matrix A .

If *uplo* = 'U', only the upper triangular part of A is used to define the elements of the symmetric matrix.

If *uplo* = 'L', only the lower triangular part of A is used to define the elements of the symmetric matrix.

ia, *ja* (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix A , respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix A .

<i>iz, jz</i>	(global) INTEGER. The row and column indices in the global array <i>z</i> indicating the first row and the first column of the submatrix <i>Z</i> , respectively.
<i>descz</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>Z</i> .
<i>work</i>	(local) REAL for pssyev. DOUBLE PRECISION for pdsyev. Array, size (<i>lwork</i>).
<i>lwork</i>	(local) INTEGER. See below for definitions of variables used to define <i>lwork</i> . If no eigenvectors are requested (<i>jobz</i> = 'N'), then <i>lwork</i> $\geq 5*n + \text{sizesytrd} + 1$, where <i>sizesytrd</i> is the workspace for p?sytrd and is $\max(\text{NB} * (np + 1), 3 * \text{NB})$. If eigenvectors are requested (<i>jobz</i> = 'V') then the amount of workspace required to guarantee that all eigenvectors are computed is: <i>qrmem</i> = $2 * n - 2$ <i>lwmin</i> = $5 * n + n * ldc + \max(\text{sizemqrleft}, \text{qrmem}) + 1$ Variable definitions: <i>nb</i> = <i>desca(mb_)</i> = <i>desca(nb_)</i> = <i>descz(mb_)</i> = <i>descz(nb_)</i> ; <i>nn</i> = $\max(n, nb, 2)$; <i>desca(rsrcc_)</i> = <i>desca(rsrcc_)</i> = <i>descz(rsrcc_)</i> = <i>descz(csrc_)</i> = 0 <i>np</i> = <i>numroc(nn, nb, 0, 0, NPROW)</i> <i>nq</i> = <i>numroc(max(n, nb, 2), nb, 0, 0, NPCOL)</i> <i>nrc</i> = <i>numroc(n, nb, myprowc, 0, NPROCS)</i> <i>ldc</i> = $\max(1, nrc)$ <i>sizemqrleft</i> is the workspace for p?ormtr when its <i>side</i> argument is 'L'. <i>myprowc</i> is defined when a new context is created as follows: call blacs_get(<i>desca(ctxt_)</i> , 0, <i>contextc</i>) call blacs_gridinit(<i>contextc</i> , 'R', NPROCS, 1) call blacs_gridinfo(<i>contextc</i> , <i>nprowc</i> , <i>npcolc</i> , <i>myprowc</i> , <i>mypcolc</i>) If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> ='L') or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is destroyed.
<i>w</i>	(global). REAL for pssyev DOUBLE PRECISION for pdsyev Array, size (<i>n</i>). On normal exit, the first <i>m</i> entries contain the selected eigenvalues in ascending order.
<i>z</i>	(local). REAL for pssyev DOUBLE PRECISION for pdsyev Array, global dimension (<i>n</i> , <i>n</i>), local dimension (<i>lld_z</i> , <i>LOCc(jz+n-1)</i>). If <i>jobz</i> = 'V', then on normal exit the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>work(1)</i>	On output, <i>work(1)</i> returns the workspace needed to guarantee completion. If the input parameters are incorrect, <i>work(1)</i> may also be incorrect. If <i>jobz</i> = 'N' <i>work(1)</i> = minimal (optimal) amount of workspace If <i>jobz</i> = 'V' <i>work(1)</i> = minimal workspace required to generate all the eigenvectors.
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . If <i>info</i> > 0: If <i>info</i> = 1 through <i>n</i> , the <i>i</i> -th eigenvalue did not converge in ?steqr2 after a total of $30n$ iterations. If <i>info</i> = <i>n</i> +1, then p?syev has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from p?syev cannot be guaranteed.

p?syevd

Computes all eigenvalues and eigenvectors of a real symmetric matrix by using a divide and conquer algorithm.

Syntax

Fortran:

```
call pssyevd(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork,
iwork, liwork, info)
```

```
call pdsyevd(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork,
iwork, liwork, info)
```

C:

```
void pssyevd (char *jobz , char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *w , float *z , MKL_INT *iz , MKL_INT *jz , MKL_INT
*descz , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT
*info );
```

```
void pdssyevd (char *jobz , char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *w , double *z , MKL_INT *iz , MKL_INT *jz , MKL_INT
*descz , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?syevd routine computes all eigenvalues and eigenvectors of a real symmetric matrix A by using a divide and conquer algorithm.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global) CHARACTER*1. Must be 'N' or 'V'.

Specifies if it is necessary to compute the eigenvectors:

If *jobz* = 'N', then only eigenvalues are computed.

If *jobz* = 'V', then eigenvalues and eigenvectors are computed.

uplo (global) CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

If *uplo* = 'U', a stores the upper triangular part of A .

If *uplo* = 'L', a stores the lower triangular part of A .

n (global) INTEGER. The number of rows and columns of the matrix A ($n \geq 0$).

a (local).

REAL for pssyevd

DOUBLE PRECISION for pdssyevd.

Block cyclic array of global dimension (n, n) and local dimension (*lld_a*, *LOCc(ja+n-1)*). On entry, the symmetric matrix A .

If *uplo* = 'U', only the upper triangular part of A is used to define the elements of the symmetric matrix.

If *uplo* = 'L', only the lower triangular part of A is used to define the elements of the symmetric matrix.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca(ctxt_)</i> is incorrect, p?syevd cannot guarantee correct error reporting.
<i>iz, jz</i>	(global) INTEGER. The row and column indices in the global array <i>z</i> indicating the first row and the first column of the submatrix <i>Z</i> , respectively.
<i>descz</i>	(global and local) INTEGER array, dimension <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> . <i>descz(ctxt_)</i> must equal <i>desca(ctxt_)</i> .
<i>work</i>	(local). REAL for pssyevd DOUBLE PRECISION for pdsyevd. Array, size <i>lwork</i> .
<i>lwork</i>	(local) INTEGER. The dimension of the array <i>work</i> . If eigenvalues are requested: $\text{lwork} \geq \max(1+6*n + 2*np*nq, \text{trilwmin}) + 2*n$ with $\text{trilwmin} = 3*n + \max(nb*(np + 1), 3*nb)$ $np = \text{numroc}(n, nb, myrow, iarow, NPROW)$ $nq = \text{numroc}(n, nb, mycol, iacol, NPCOL)$ If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by pxerbla.
<i>iwork</i>	(local) INTEGER. Workspace array, dimension <i>liwork</i> .
<i>liwork</i>	(local) INTEGER, dimension of <i>iwork</i> . $\text{liwork} = 7*n + 8*npcol + 2.$

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L'), or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>w</i>	(global). REAL for pssyevd DOUBLE PRECISION for pdsyevd. Array, size <i>n</i> . If <i>info</i> = 0, <i>w</i> contains the eigenvalues in the ascending order.
<i>z</i>	(local).

REAL for pssyevd
 DOUBLE PRECISION for pdsyevd.
 Array, global dimension (n, n), local dimension ($lld_z, LOCo(jz+n-1)$).
 The z parameter contains the orthonormal eigenvectors of the matrix A .

work(1) On exit, returns adequate workspace to allow optimal performance.

iwork(1) (local).
 On exit, if $liwork > 0$, *iwork(1)* returns the optimal *liwork*.

info (global) INTEGER.
 If $info = 0$, the execution is successful.
 If $info < 0$:
 If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$. If the i -th argument is a scalar and had an illegal value, then $info = -i$.
 If $info > 0$:
 The algorithm failed to compute the $info/(n+1)$ -th eigenvalue while working on the submatrix lying in global rows and columns $\text{mod}(info, n+1)$.

p?syevr

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using Relatively Robust Representation.

Syntax

Fortran:

```
call pssyevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z,  

  iz, jz, descz, work, lwork, iwork, liwork, info )
call pdsyevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z,  

  iz, jz, descz, work, lwork, iwork, liwork, info )
```

Include Files

- C: mkl_scalapack.h

Description

p?syevr computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A distributed in 2D blockcyclic format by calling the recommended sequence of ScaLAPACK routines.

First, the matrix A is reduced to real symmetric tridiagonal form. Then, the eigenproblem is solved using the parallel MRRR algorithm. Last, if eigenvectors have been computed, a backtransformation is done.

Upon successful completion, each processor stores a copy of all computed eigenvalues in w . The eigenvector matrix z is stored in 2D block-cyclic format distributed over all processors.

Note that subsets of eigenvalues/vectors can be selected by specifying a range of values or a range of indices for the desired eigenvalues.

Input Parameters

<i>jobz</i>	(global) CHARACTER*1
	Specifies whether or not to compute the eigenvectors: = 'N': Compute eigenvalues only. = 'V': Compute eigenvalues and eigenvectors.
<i>range</i>	(global) CHARACTER*1
	= 'A': all eigenvalues will be found. = 'V': all eigenvalues in the interval [<i>vl</i> , <i>vu</i>] will be found. = 'I': the <i>il</i> -th through <i>iu</i> -th eigenvalues will be found.
<i>uplo</i>	(global) CHARACTER*1
	Specifies whether the upper or lower triangular part of the symmetric matrix <i>A</i> is stored: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	(global) INTEGER
	The number of rows and columns of the matrix <i>a</i> . <i>n</i> ≥ 0
<i>a</i>	REAL for pssyevr DOUBLE PRECISION for pdsyevr
	Block cyclic array of global dimension (<i>n</i> , <i>n</i>), local dimension (<i>lld_a</i> , <i>LOC_c(ja+n-1)</i>).
	This array contains the local pieces of the symmetric distributed matrix <i>A</i> . If <i>uplo</i> = 'U', only the upper triangular part of <i>a</i> is used to define the elements of the symmetric matrix. If <i>uplo</i> = 'L', only the lower triangular part of <i>a</i> is used to define the elements of the symmetric matrix.
	On exit, the lower triangle (if <i>uplo</i> ='L') or the upper triangle (if <i>uplo</i> ='U') of <i>a</i> , including the diagonal, is destroyed.
<i>ia</i>	(global) INTEGER
	Global row index of <i>a</i> , which points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>ja</i>	(global) INTEGER
	Global column index of <i>a</i> , which points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>desca</i>	(global and local) INTEGER array of dimension <i>dlen_=9</i> .
	The array descriptor for the distributed matrix <i>a</i> .
<i>vl</i>	REAL for pssyevr DOUBLE PRECISION for pdsyevr

(global)

If *range*='V', the lower bound of the interval to be searched for eigenvalues. Not referenced if *range* = 'A' or 'I'.

vu

REAL for pssyevr

DOUBLE PRECISION for pdsyevr

(global)

If *range*='V', the upper bound of the interval to be searched for eigenvalues. Not referenced if *range* = 'A' or 'I'.

il

(global) INTEGER

If *range*='I', the index (from smallest to largest) of the smallest eigenvalue to be returned. $il \geq 1$.

Not referenced if *range* = 'A'.

iu

(global) INTEGER

If *range*='I', the index (from smallest to largest) of the largest eigenvalue to be returned. $\min(il,n) \leq iu \leq n$.

Not referenced if *range* = 'A'.

iz

(global) INTEGER

Global row index of *z*, which points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

rz

(global) INTEGER

Global column index of *z*, which points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

descz

() INTEGER array of dimension *dlen_*.

The array descriptor for the distributed matrix *z*.

The context *descz(ctxt_)* must equal *desca(ctxt_)*. Also note the array alignment requirements specified below.

work

REAL for pssyevr

DOUBLE PRECISION for pdsyevr

(local workspace) array, dimension (*lwork*)

lwork

(local) INTEGER

Size of *work*, must be at least 3.

See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N') then

$lwork \geq 2 + 5*n + \max(12 * nn, neig * (np0 + 1))$

If eigenvectors are requested (*jobz* = 'V') then the amount of workspace required is:

$lwork \geq 2 + 5*n + \max(18*nn, np0 * mq0 + 2 * neig * neig) + (2 + \text{ceil}(neig, nprow*ncol))*nn$

Variable definitions:

$neig$ = number of eigenvectors requested

nb = $\text{desc}(mb_{_}) = \text{desc}(nb_{_}) = \text{descz}(mb_{_}) = \text{descz}(nb_{_})$

$nn = \max(n, neig, 2)$

$\text{desc}(rsrc_{_}) = \text{desc}(csrc_nb_{_}) = \text{descz}(rsrc_{_}) = \text{descz}(csrc_{_}) = 0$

$np0 = \text{numroc}(nn, neig, 0, 0, nprow)$

$mq0 = \text{numroc}(\max(neig, neig, 2), neig, 0, 0, npcol)$

$\text{ceil}(x, y)$ is a ScaLAPACK function returning $\text{ceiling}(x/y)$, and $nprow$ and $npcol$ can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pixerbla`.

$liwork$ (local) INTEGER

size of $iwork$

Let $nnp = \max(n, nprow*ncol + 1, 4)$. Then:

$liwork \geq 12*nnp + 2*n$ when the eigenvectors are desired

$liwork \geq 10*nnp + 2*n$ when only the eigenvalues have to be computed

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pixerbla`.

OUTPUT Parameters

m (global) INTEGER

Total number of eigenvalues found. $0 \leq m \leq n$.

nz (global) INTEGER

Total number of eigenvectors computed. $0 \leq nz \leq m$.

The number of columns of z that are filled.

If $jobz \neq 'V'$, nz is not referenced.

If $jobz = 'V'$, $nz = m$

w REAL for `pssyevr`

DOUBLE PRECISION for `pdsyevr`

(global) array, dimension (n)

Upon successful exit, the first m entries contain the selected eigenvalues in ascending order.

<i>z</i>	REAL for pssyevr DOUBLE PRECISION for pdsyevr Block-cyclic array, global dimension (<i>n</i> , <i>n</i>), local dimension (<i>lld_z</i> , <i>LOC_c(jz+n-1)</i>) On exit, contains local pieces of distributed matrix <i>Z</i> .
<i>work</i>	On return, <i>work</i> (1) contains the optimal amount of workspace required for efficient execution. If <i>jobz</i> ='N' <i>work</i> (1) = optimal amount of workspace required to compute the eigenvalues. If <i>jobz</i> ='V' <i>work</i> (1) = optimal amount of workspace required to compute eigenvalues and eigenvectors.
<i>iwork</i>	(local workspace) INTEGER array On return, <i>iwork</i> (1) contains the amount of integer workspace required.
<i>info</i>	(global) INTEGER = 0: successful exit < 0: If the <i>i</i> -th argument is an array and the <i>j</i> th-entry had an illegal value, then <i>info</i> = -(<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

Application Notes

The distributed submatrices *a*(*ia*:*, *ja*:*) and *z*(*iz*:*iz*+*m*-1, *iz*:*iz*+*n*-1) must satisfy the following alignment properties:

1. Identical (quadratic) dimension: *desc_a*(*m*_)=*desc_z*(*m*_)=*desc_a*(*n*_)=*desc_z*(*n*_)
2. Quadratic conformal blocking: *desc_a*(*mb*_)=*desc_a*(*nb*_)=*desc_z*(*mb*_)=*desc_z*(*nb*_), *desc_a*(*rsrc*_)=*desc_z*(*rsrc*_)
3. $\text{mod}(\text{ia}-1, \text{mb_a}) = \text{mod}(\text{iz}-1, \text{mb_z}) = 0$

p?syevx

Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

Syntax

Fortran:

```
call pssyevx(jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr, gap, info)
call pdsyevx(jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork, ifail, iclustr, gap, info)
```

C:

```
void pssyevx (char *jobz , char *range , char *uplo , MKL_INT *n , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *vl , float *vu , MKL_INT *il , MKL_INT
*iu , float *abstol , MKL_INT *m , MKL_INT *nz , float *w , float *orfac , float *z ,
MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , float *work , MKL_INT *lwork , MKL_INT
*iwork , MKL_INT *liwork , MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT
*info );
```

```
void pdssyevx (char *jobz , char *range , char *uplo , MKL_INT *n , double *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , double *vl , double *vu , MKL_INT *il , MKL_INT
*iu , double *abstol , MKL_INT *m , MKL_INT *nz , double *w , double *orfac , double
*z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , double *work , MKL_INT *lwork ,
MKL_INT *iwork , MKL_INT *liwork , MKL_INT *ifail , MKL_INT *iclustr , double *gap ,
MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?syevx` routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A by calling the recommended sequence of ScalAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

jobz (global) CHARACTER*1. Must be '`N`' or '`V`'. Specifies if it is necessary to compute the eigenvectors:

If *jobz* = '`N`', then only eigenvalues are computed.

If *jobz* = '`V`', then eigenvalues and eigenvectors are computed.

range (global) CHARACTER*1. Must be '`A`', '`V`', or '`I`'.

If *range* = '`A`', all eigenvalues will be found.

If *range* = '`V`', all eigenvalues in the half-open interval $[vl, vu]$ will be found.

If *range* = '`I`', the eigenvalues with indices *il* through *iu* will be found.

uplo (global) CHARACTER*1. Must be '`U`' or '`L`'.

Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:

If *uplo* = '`U`', *a* stores the upper triangular part of A .

If *uplo* = '`L`', *a* stores the lower triangular part of A .

n (global) INTEGER. The number of rows and columns of the matrix A ($n \geq 0$).

a (local). REAL for `pdssyevx`

DOUBLE PRECISION for `pdssyevx`.

Block cyclic array of global dimension (n, n) and local dimension $(lld_a, LOCc(ja+n-1))$. On entry, the symmetric matrix A .

If *uplo* = '`U`', only the upper triangular part of A is used to define the elements of the symmetric matrix.

If *uplo* = '`L`', only the lower triangular part of A is used to define the elements of the symmetric matrix.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>vl, vu</i>	(global) REAL for pssyevx DOUBLE PRECISION for pdssyevx. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; <i>vl</i> ≤ <i>vu</i> . Not referenced if <i>range</i> = 'A' or 'I'.
<i>il, iu</i>	(global) INTEGER. If <i>range</i> = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints: <i>il</i> ≥ 1 $\min(il, n) \leq iu \leq n$ Not referenced if <i>range</i> = 'A' or 'V'.
<i>abstol</i>	(global). REAL for pssyevx DOUBLE PRECISION for pdssyevx. If <i>jobz</i> ='V', setting <i>abstol</i> to <i>p?lamch(context, 'U')</i> yields the most orthogonal eigenvectors. The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + eps * \max(a , b),$ where <i>eps</i> is the machine precision. If <i>abstol</i> is less than or equal to zero, then <i>eps*norm(T)</i> will be used in its place, where <i>norm(T)</i> is the 1-norm of the tridiagonal matrix obtained by reducing <i>A</i> to tridiagonal form. Eigenvalues will be computed most accurately when <i>abstol</i> is set to twice the underflow threshold $2*p?lamch('S')$ not zero. If this routine returns with $((\text{mod}(info, 2).ne.0).or. * (\text{mod}(info/8, 2).ne.0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting <i>abstol</i> to $2*p?lamch('S')$.
<i>orfac</i>	(global). REAL for pssyevx DOUBLE PRECISION for pdssyevx. Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within <i>tol=orfac*norm(A)</i> of each other are to be reorthogonalized. However, if the workspace is insufficient (see <i>lwork</i>), <i>tol</i> may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if <i>orfac</i> equals zero. A default value of 1.0e-3 is used if <i>orfac</i> is negative. <i>orfac</i> should be identical on all processes.

iz, jz

(global) INTEGER. The row and column indices in the global array *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*. *descz(ctxt_)* must equal *desca(ctxt_)*.

work

(local)

REAL for pssyevx.

DOUBLE PRECISION for pdsyevx.

Array, size (*lwork*).

lwork

(local) INTEGER. The dimension of the array *work*.

See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then *lwork* $\geq 5*n + \max(5*nn, NB*(np0 + 1))$.

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

lwork $\geq 5*n + \max(5*nn, np0*mq0 + 2*NB*NB) + \text{ceil}(neig, \text{NPROW}*\text{NPCOL}) * nn$

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following to *lwork*:

*(clustersize-1)*n*,

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

{*w(k), ..., w(k+clustersize-1)* | *w(j+1) ≤ w(j)* + *orfac*2*norm(A)*},

where

neig = number of eigenvectors requested

nb = *desca(mb_)* = *desca(nb_)* = *descz(mb_)* = *descz(nb_)*;

nn = *max(n, nb, 2)*;

desca(rsra_) = *desca(nb_)* = *descz(rsra_)* = *descz(csrc_)* = 0;

np0 = *numroc(nn, nb, 0, 0, NPROW)*;

mq0 = *numroc(max(neig, nb, 2), nb, 0, 0, NPCOL)*

ceil(x, y) is a ScaLAPACK function returning ceiling(*x/y*)

If *lwork* is too small to guarantee orthogonality, p?syevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.

Note that when `range='V'`, number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if `lwork` is large enough to compute the eigenvalues, `p?sygvx` computes the eigenvalues and as many eigenvectors as possible.

Relationship between workspace, orthogonality & performance:

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

`lwork ≥ max(lwork, 5*n + nsytrd_lwopt),`

where `lwork`, as defined previously, depends upon the number of eigenvectors requested, and

```
nsytrd_lwopt = n + 2*(anb+1)*(4*nps+2) + (nps + 3)*nps;
anb = pjlaenv(desca(ctxt_), 3, 'p?syttrd', 'L', 0, 0, 0, 0);
sqnpc = int(sqrt(dble(NPROW * NPCOL)));
nps = max(numroc(n, 1, 0, 0, sqnpc), 2*anb);
numroc is a ScaLAPACK tool functions;
pjlaenv is a ScaLAPACK environmental inquiry function
NPROW, MYCOL, NPROW and NPCOL can be determined by calling the
subroutine blacs_gridinfo.
```

For large `n`, no extra workspace is needed, however the biggest boost in performance comes for small `n`, so it is wise to provide the extra workspace (typically less than a megabyte per process).

If `clustersize > n/sqrt(NPROW*NPCOL)`, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, `clustersize = n-1`) `p?stein` will perform no better than `?stein` on single processor.

For `clustersize = n/sqrt(NPROW*NPCOL)` reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For `clustersize > n/sqrt(NPROW*NPCOL)` execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pixerbla`.

`iwork`

(local) INTEGER. Workspace array.

`liwork`

(local) INTEGER, dimension of `iwork`. `liwork ≥ 6*nnp`

Where: `nnp = max(n, NPROW*NPCOL + 1, 4)`

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

- a
On exit, the lower triangle (if $uplo = 'L'$) or the upper triangle (if $uplo = 'U'$) of A , including the diagonal, is overwritten.
- m
(global) INTEGER. The total number of eigenvalues found; $0 \leq m \leq n$.
- nz
(global) INTEGER. Total number of eigenvectors computed. $0 \leq nz \leq m$.
The number of columns of z that are filled.
If $jobz \neq 'V'$, nz is not referenced.
If $jobz = 'V'$, $nz = m$ unless the user supplies insufficient space and `p?syevx` is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in z ($m.1e.descz(n_)$) and sufficient workspace to compute them. (See $lwork$). `p?syevx` is always able to detect insufficient space without computation unless `range.eq.'V'`.
- w
(global). REAL for `pssyevx`
DOUBLE PRECISION for `pdsyevx`.
Array, size (n). The first m elements contain the selected eigenvalues in ascending order.
- z
(local). REAL for `pssyevx`
DOUBLE PRECISION for `pdsyevx`.
Array, global dimension (n, n), local dimension ($lld_z, LOCC(jz+n-1)$).
If $jobz = 'V'$, then on normal exit the first m columns of z contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in `ifail`.
If $jobz = 'N'$, then z is not referenced.
- $work(1)$
On exit, returns workspace adequate workspace to allow optimal performance.
- $iwork(1)$
On return, $iwork(1)$ contains the amount of integer workspace required.
- $ifail$
(global) INTEGER.
Array, size (n).
If $jobz = 'V'$, then on normal exit, the first m elements of $ifail$ are zero. If $(\text{mod}(info, 2) . ne. 0)$ on exit, then $ifail$ contains the indices of the eigenvectors that failed to converge.
If $jobz = 'N'$, then $ifail$ is not referenced.

iclustr

(global) INTEGER. Array, size (2*NPROW*NPCOL)

This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*($2*i-1$) to *iclustr*($2*i$), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr()* is a zero terminated array. (*iclustr*($2*k$).ne.0. and. *iclustr*($2*k+1$).eq.0) if and only if k is the number of clusters.

iclustr is not referenced if *jobz* = 'N'.

gap

(global)

REAL for pssyevx

DOUBLE PRECISION for pdsyevx.

Array, size (NPROW*NPCOL)

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the i -th cluster may be as high as $(C*n) / gap(i)$ where C is a small constant.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0:

If the i -th argument is an array and the j -entry had an illegal value, then *info* = $-(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then *info* = $-i$.

If *info* > 0: if (mod(*info*, 2).ne.0), then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure *abstol*=2.0*p?lamch('U').

If (mod(*info*/2, 2).ne.0), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If (mod(*info*/4, 2).ne.0), then space limit prevented p?syevf from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If (mod(*info*/8, 2).ne.0), then p?stebz failed to compute eigenvalues. Ensure *abstol*=2.0*p?lamch('U').

p?heev

Computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix.

Syntax

Fortran:

```
call pcheev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, rwork,
lrwork, info)

call pzheev(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork, rwork,
lrwork, info)
```

C:

```
void pcheev (char *jobz , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *w , MKL_Complex8 *z , MKL_INT *iz , MKL_INT
*jz , MKL_INT *descz , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT
*lrwork , MKL_INT *info );

void pzheev (char *jobz , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *w , MKL_Complex16 *z , MKL_INT *iz , MKL_INT
*jz , MKL_INT *descz , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT
*lrwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?heev` routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A by calling the recommended sequence of ScaLAPACK routines. The routine assumes a homogeneous system and makes spot checks of the consistency of the eigenvalues across the different processes. A heterogeneous system may return incorrect results without any error messages.

Input Parameters

`np` = the number of rows local to a given process.

`nq` = the number of columns local to a given process.

`jobz` (global) CHARACTER*1. Must be '`N`' or '`V`'.

Specifies if it is necessary to compute the eigenvectors:

If `jobz` = '`N`', then only eigenvalues are computed.

If `jobz` = '`V`', then eigenvalues and eigenvectors are computed.

`uplo` (global) CHARACTER*1. Must be '`U`' or '`L`'.

Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

If `uplo` = '`U`', a stores the upper triangular part of A .

If `uplo` = '`L`', a stores the lower triangular part of A .

`n` (global) INTEGER. The number of rows and columns of the matrix A ($n \geq 0$).

`a` (local).

COMPLEX for pcheev

DOUBLE COMPLEX for pzheev.

Block cyclic array of global dimension (n, n) and local dimension $(lld_a,$ $LOCc(ja+n-1))$. On entry, the Hermitian matrix A .

If $uplo = 'U'$, only the upper triangular part of A is used to define the elements of the Hermitian matrix.

If $uplo = 'L'$, only the lower triangular part of A is used to define the elements of the Hermitian matrix.

ia, ja

(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

$desc_a$

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A . If $desc_a(ctxt_)$ is incorrect, p?heev cannot guarantee correct error reporting.

iz, jz

(global) INTEGER. The row and column indices in the global array z indicating the first row and the first column of the submatrix Z , respectively.

$desc_z$

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix Z . $desc_z(ctxt_)$ must equal $desc_a(ctxt_)$.

$work$

(local).

COMPLEX for pcheev

DOUBLE COMPLEX for pzheev.

Array, size $lwork$.

$lwork$

(local) INTEGER. The dimension of the array $work$.

If only eigenvalues are requested ($jobz = 'N'$):

$lwork \geq \max(nb * (np0 + 1), 3) + 3*n$

If eigenvectors are requested ($jobz = 'V'$), then the amount of workspace required:

$lwork \geq (np0 + nq0 + nb) * nb + 3*n + n^2$

with $nb = desc_a(mb_) = desc_a(nb_) = nb = desc_z(mb_) = desc_z(nb_)$

$np0 = numroc(nn, nb, 0, 0, NPROW)$.

$nq0 = numroc(\max(n, nb, 2), nb, 0, 0, NPCOL)$.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by pxerbla.

$rwork$

(local).

REAL for pcheev

DOUBLE PRECISION for pzheev.

Workspace array, size $lrwork$.

$lrwork$

(local) INTEGER. The dimension of the array $rwork$.

See below for definitions of variables used to define *lrwork*.

If no eigenvectors are requested (*jobz* = 'N'), then *lrwork* $\geq 2*n$.

If eigenvectors are requested (*jobz* = 'V'), then *lrwork* $\geq 2*n + 2*n-2$.

If *lrwork* = -1, then *lrwork* is global input and a workspace query is assumed; the routine only calculates the minimum size required for the *rwork* array. The required workspace is returned as the first element of *rwork*, and no error message is issued by *pixerbla*.

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L'), or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>w</i>	(global). REAL for <i>pcheev</i> DOUBLE PRECISION for <i>pzheev</i> . Array, size <i>n</i> . The first <i>m</i> elements contain the selected eigenvalues in ascending order.
<i>z</i>	(local). COMPLEX for <i>pcheev</i> DOUBLE COMPLEX for <i>pzheev</i> . Array, global dimension (<i>n</i> , <i>n</i>), local dimension (<i>lld_z</i> , <i>LOCc(jz+n-1)</i>). If <i>jobz</i> = 'V', then on normal exit the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>work(1)</i>	On exit, returns adequate workspace to allow optimal performance. If <i>jobz</i> = 'N', then <i>work(1)</i> = minimal workspace only for eigenvalues. If <i>jobz</i> = 'V', then <i>work(1)</i> = minimal workspace required to generate all the eigenvectors.
<i>rwork(1)</i>	(local) COMPLEX for <i>pcheev</i> DOUBLE COMPLEX for <i>pzheev</i> . On output, <i>rwork(1)</i> returns workspace required to guarantee completion.
<i>info</i>	(global) INTEGER. If <i>info</i> = 0, the execution is successful. If <i>info</i> < 0:

If the i -th argument is an array and the j -entry had an illegal value, then $\text{info} = -(i*100+j)$. If the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.

If $\text{info} > 0$:

If $\text{info} = 1$ through n , the i -th eigenvalue did not converge in [?steqr2](#) after a total of $30*n$ iterations.

If $\text{info} = n+1$, then p?heev detected heterogeneity, and the accuracy of the results cannot be guaranteed.

p?heevd

Computes all eigenvalues and eigenvectors of a complex Hermitian matrix by using a divide and conquer algorithm.

Syntax

Fortran:

```
call pccheevd(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork,
rwork, lrwork, iwork, liwork, info)
```

```
call pzcheevd(jobz, uplo, n, a, ia, ja, desca, w, z, iz, jz, descz, work, lwork,
rwork, lrwork, iwork, liwork, info)
```

C:

```
void pccheevd (char *jobz , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *w , MKL_Complex8 *z , MKL_INT *iz , MKL_INT
*jz , MKL_INT *descz , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT
*lrwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );

void pzcheevd (char *jobz , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *w , MKL_Complex16 *z , MKL_INT *iz , MKL_INT
*jz , MKL_INT *descz , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT
*lrwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?heevd routine computes all eigenvalues and eigenvectors of a complex Hermitian matrix A by using a divide and conquer algorithm.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

$jobz$ (global) CHARACTER*1. Must be 'N' or 'V'.

Specifies if it is necessary to compute the eigenvectors:

If $jobz = 'N'$, then only eigenvalues are computed.

If $jobz = 'V'$, then eigenvalues and eigenvectors are computed.

uplo (global) CHARACTER*1. Must be 'U' or 'L'.
 Specifies whether the upper or lower triangular part of the Hermitian matrix *A* is stored:
 If *uplo* = 'U', *a* stores the upper triangular part of *A*.
 If *uplo* = 'L', *a* stores the lower triangular part of *A*.

n (global) INTEGER. The number of rows and columns of the matrix *A* (*n* \geq 0).

a (local).
 COMPLEX for pcheevd
 DOUBLE COMPLEX for pzheevd.
 Block cyclic array of global dimension (*n*, *n*) and local dimension (*lld_a*, *LOCc(ja+n-1)*). On entry, the Hermitian matrix *A*.
 If *uplo* = 'U', only the upper triangular part of *A* is used to define the elements of the Hermitian matrix.
 If *uplo* = 'L', only the lower triangular part of *A* is used to define the elements of the Hermitian matrix.

ia, ja (global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca (global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *A*. If *desca(ctxt_)* is incorrect, p?heevd cannot guarantee correct error reporting.

iz, jz (global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, dimension *dlen_*. The array descriptor for the distributed matrix *Z*. *descz(ctxt_)* must equal *desca(ctxt_)*.

work (local).
 COMPLEX for pcheevd
 DOUBLE COMPLEX for pzheevd.
 Array, size *lwork*.

lwork (local) INTEGER. The dimension of the array *work*.
 If eigenvalues are requested:

$$\textit{lwork} = n + (\textit{nb0} + \textit{mq0} + \textit{nb}) * \textit{nb}$$

$$\text{with } \textit{np0} = \text{numroc}(\max(n, nb, 2), nb, 0, 0, \text{NPROW})$$

$$\textit{mq0} = \text{numroc}(\max(n, nb, 2), nb, 0, 0, \text{NPCOL})$$

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by pxerbla.

 $rwork$

(local).

REAL for pcheevd

DOUBLE PRECISION for pzheevd.

Workspace array, size $lrwork$. $lrwork$ (local) INTEGER. The dimension of the array $rwork$. $lrwork \geq 1 + 9*n + 3*np*nq,$ with $np = \text{numroc}(n, nb, myrow, irow, \text{NPROW})$ $nq = \text{numroc}(n, nb, mycol, iacol, \text{NPCOL})$ $iwork$ (local) INTEGER. Workspace array, dimension $liwork$. $liwork$ (local) INTEGER, dimension of $iwork$. $liwork = 7*n + 8*ncol + 2.$

Output Parameters

 a On exit, the lower triangle (if $uplo = 'L'$), or the upper triangle (if $uplo = 'U'$) of A , including the diagonal, is overwritten. w

(global).

REAL for pcheevd

DOUBLE PRECISION for pzheevd.

Array, size n . If $info = 0$, w contains the eigenvalues in the ascending order. z

(local).

COMPLEX for pcheevd

DOUBLE COMPLEX for pzheevd.

Array, global dimension (n, n) , local dimension $(lld_z, \text{LOCc}(jz+n-1))$.The z parameter contains the orthonormal eigenvectors of the matrix A . $work(1)$

On exit, returns adequate workspace to allow optimal performance.

 $rwork(1)$

(local)

COMPLEX for pcheevd

DOUBLE COMPLEX for pzheevd.

On output, $rwork(1)$ returns workspace required to guarantee completion. $iwork(1)$

(local).

On return, *iwork*(1) contains the amount of integer workspace required.

info (global) INTEGER.
 If *info* = 0, the execution is successful.
 If *info* < 0:
 If the *i*-th argument is an array and the *j*-entry had an illegal value, then
 info = -(*i**100+*j*). If the *i*-th argument is a scalar and had an illegal
 value, then *info* = -*i*.
 If *info* > 0:
 If *info* = 1 through *n*, the *i*-th eigenvalue did not converge.

p?heevr

Computes selected eigenvalues and, optionally,
 eigenvectors of a Hermitian matrix using Relatively
 Robust Representation.

Syntax

Fortran:

```
call pccheevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z,
               iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, info )
call pzheevr( jobz, range, uplo, n, a, ia, ja, desca, vl, vu, il, iu, m, nz, w, z,
               iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, info )
```

Include Files

- C: mkl_scalapack.h

Description

p?heevr computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix *A* distributed in 2D blockcyclic format by calling the recommended sequence of ScalAPACK routines.

First, the matrix *A* is reduced to complex Hermitian tridiagonal form. Then, the eigenproblem is solved using the parallel MRRR algorithm. Last, if eigenvectors have been computed, a backtransformation is done.

Upon successful completion, each processor stores a copy of all computed eigenvalues in *w*. The eigenvector matrix *Z* is stored in 2D block-cyclic format distributed over all processors.

Input Parameters

<i>jobz</i>	(global) CHARACTER*1
	Specifies whether or not to compute the eigenvectors:
	= 'N': Compute eigenvalues only.
	= 'V': Compute eigenvalues and eigenvectors.
<i>range</i>	(global) CHARACTER*1
	= 'A': all eigenvalues will be found.
	= 'V': all eigenvalues in the interval [vl,vu] will be found.
	= 'I': the <i>il</i> -th through <i>iu</i> -th eigenvalues will be found.

<i>uplo</i>	(global) CHARACTER*1
	Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is stored: = 'U': Upper triangular = 'L': Lower triangular
<i>n</i>	(global) INTEGER
	The number of rows and columns of the matrix <i>A</i> . $n \geq 0$
<i>a</i>	COMPLEX for pcheevr COMPLEX*16 for pzheevr Block-cyclic array, global dimension (<i>n</i> , <i>n</i>), local dimension (<i>lld_a</i> , <i>LOC_c(ja+n-1)</i>) Contains the local pieces of the Hermitian distributed matrix <i>A</i> . If <i>uplo</i> = 'U', only the upper triangular part of <i>a</i> is used to define the elements of the Hermitian matrix. If <i>uplo</i> = 'L', only the lower triangular part of <i>a</i> is used to define the elements of the Hermitian matrix.
<i>ia</i>	(global) INTEGER Global row index of <i>a</i> , which points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>ja</i>	(global) INTEGER Global column index of <i>a</i> , which points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
<i>desca</i>	(global and local) INTEGER array of dimension <i>dlen_</i> . (The ScaLAPACK descriptor length is <i>dlen_</i> = 9.) The array descriptor for the distributed matrix <i>a</i> . The descriptor stores details about the 2D block-cyclic storage, see the notes below. If <i>desca</i> is incorrect, p?heevr cannot work correctly. Also note the array alignment requirements specified below
<i>vl</i>	REAL for pcheevr DOUBLE PRECISION for pzheevr (global) If <i>range</i> ='V', the lower bound of the interval to be searched for eigenvalues. Not referenced if <i>range</i> = 'A' or 'I'.
<i i="" vu<=""></i>	REAL for pcheevr DOUBLE PRECISION for pzheevr (global) If <i>range</i> ='V', the upper bound of the interval to be searched for eigenvalues. Not referenced if <i>range</i> = 'A' or 'I'.
<i>il</i>	(global) INTEGER

If $\text{range} = \text{'I}'$, the index (from smallest to largest) of the smallest eigenvalue to be returned. $\text{il} \geq 1$.

Not referenced if $\text{range} = \text{'A'}$.

iu

(global) INTEGER

If $\text{range} = \text{'I}'$, the index (from smallest to largest) of the largest eigenvalue to be returned. $\min(\text{il}, n) \leq \text{iu} \leq n$.

Not referenced if $\text{range} = \text{'A'}$.

iz

(global) INTEGER

Global row index of z , which points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

jz

(global) INTEGER

Global column index of z , which points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

descz

(global and local) INTEGER array of dimension dlen_z .

The array descriptor for the distributed matrix z . $\text{descz}(\text{ctxt}_\text{z})$ must equal $\text{desca}(\text{ctxt}_\text{z})$

work

COMPLEX for pccheevr

COMPLEX*16 for pzheevr

(local workspace) array, dimension (lwork)

lwork

(local) INTEGER

Size of work array, must be at least 3.

If only eigenvalues are requested:

$$\text{lwork} \geq n + \max(n * (\text{np00} + 1), n * 3)$$

If eigenvectors are requested:

$$\text{lwork} \geq n + (\text{np00} + \text{mq00} + n) * n$$

For definitions of np00 and mq00 , see lrwork .

For optimal performance, greater workspace is needed, i.e.

$$\text{lwork} \geq \max(\text{lwork}, \text{nhetrd_lwork})$$

Where lwork is as defined above, and

$$\text{nhetrd_lwork} = n + 2 * (\text{anb} + 1) * (4 * \text{nps} + 2) + (\text{nps} + 1) * \text{nps}$$

$$\text{ictxt} = \text{desca}(\text{ctxt}_\text{z})$$

$$\text{anb} = \text{pjlaenv}(\text{ictxt}, 3, \text{'PCHEATTRD'}, \text{'L'}, 0, 0, 0, 0, 0)$$

$$\text{sqnpc} = \sqrt{\text{real}(\text{nprow} * \text{npcol})}$$

$$\text{nps} = \max(\text{numroc}(n, 1, 0, 0, \text{sqnpc}), 2 * \text{anb})$$

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

$rwork$

REAL for `pcheevr`

DOUBLE PRECISION for `pzheevr`

(local workspace) array, dimension ($lrwork$)

$lrwork$

(local) INTEGER

Size of $rwork$, must be at least 3.

See below for definitions of variables used to define $lrwork$.

If no eigenvectors are requested ($jobz = 'N'$) then

$$lrwork \geq 2 + 5 * n + \max(12 * n, nb * (np00 + 1))$$

If eigenvectors are requested ($jobz = 'V'$) then the amount of workspace required is:

$$\begin{aligned} lrwork \geq & 2 + 5 * n + \max(18 * n, np00 * mq00 + 2 * nb * nb) + \\ & (2 + \text{ceil}(neig, nprow * ncol)) * n \end{aligned}$$

Variable definitions:

$neig$ = number of eigenvectors requested

nb = $\text{desc}(mb_{_}) = \text{desc}(nb_{_}) = \text{desc}(mb_{_}) = \text{desc}(nb_{_})$

$nn = \max(n, nb, 2)$

$\text{desc}(rsrc_{_}) = \text{desc}(csrc_{_}) = \text{desc}(rsrc_{_}) = \text{desc}(csrc_{_}) = 0$

$np00 = \text{numroc}(nn, nb, 0, 0, nprow)$

$mq00 = \text{numroc}(\max(neig, nb, 2), nb, 0, 0, ncol)$

$\text{ceil}(x, y)$ is a ScaLAPACK function returning ceiling(x/y), and $nprow$ and $ncol$ can be determined by calling the subroutine `blacs_gridinfo`.

If $lrwork = -1$, then $lrwork$ is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by [pxerbla](#)

$iwork$

(local workspace) INTEGER array, dimension ($liwork$)

$liwork$

(local) INTEGER

size of $iwork$

Let $nnp = \max(n, nprow * ncol + 1, 4)$. Then:

$liwork \geq 12 * nnp + 2 * n$ when the eigenvectors are desired

$liwork \geq 10 * nnp + 2 * n$ when only the eigenvalues have to be computed

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pixerbla`

OUTPUT Parameters

a	The lower triangle (if $uplo='L'$) or the upper triangle (if $uplo='U'$) of a , including the diagonal, is destroyed.
m	(global) INTEGER Total number of eigenvalues found. $0 \leq m \leq n$.
nz	(global) INTEGER Total number of eigenvectors computed. $0 \leq nz \leq m$. The number of columns of z that are filled. If $jobz \neq 'V'$, nz is not referenced. If $jobz = 'V'$, $nz = m$
w	REAL for <code>pcheevr</code> DOUBLE PRECISION for <code>pzheevr</code> (global) array, dimension (n) On normal exit, the first m entries contain the selected eigenvalues in ascending order.
z	COMPLEX for <code>pcheevr</code> COMPLEX*16 for <code>pzheevr</code> (local) array, global dimension (n, n), local dimension ($lld_z, LOC_c(jz +n-1)$) If $jobz = 'V'$, then on normal exit the first m columns of z contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If $jobz = 'N'$, then z is not referenced.
$work$	$work(1)$ returns workspace adequate workspace to allow optimal performance.
$rwork$	On return, $rwork(1)$ contains the optimal amount of workspace required for efficient execution. if $jobz='N'$ $rwork(1) =$ optimal amount of workspace required to compute the eigenvalues. if $jobz='V'$ $rwork(1) =$ optimal amount of workspace required to compute eigenvalues and eigenvectors.
$iwork$	On return, $iwork(1)$ contains the amount of integer workspace required.
$info$	(global) INTEGER = 0: successful exit

< 0: If the i -th argument is an array and the j -th entry had an illegal value, then $\text{info} = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $\text{info} = -i$.

Application Notes

The distributed submatrices $a(ia:*, ja:*)$ and $z(iz:iz+m-1, jz:jz+n-1)$ must satisfy the following alignment properties:

1. Identical (quadratic) dimension: $\text{desc}(m_*) = \text{desc}(m_*) = \text{desc}(n_*) = \text{desc}(n_*)$
2. Quadratic conformal blocking: $\text{desc}(mb_*) = \text{desc}(nb_*) = \text{desc}(mb_*) = \text{desc}(nb_*)$, $\text{desc}(rsrc_*) = \text{desc}(rsrc_*)$
3. $\text{mod}(ia-1, mb_a) = \text{mod}(iz-1, mb_z) = 0$

p?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

Fortran:

```
call pccheevx(jobz, range, uplo, n, a, ia, ja, desc, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, ifail, iclustr, gap, info)
call pzheevx(jobz, range, uplo, n, a, ia, ja, desc, vl, vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork, iwork, liwork, ifail, iclustr, gap, info)
```

C:

```
void pccheevx (char *jobz , char *range , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc , float *vl , float *vu , MKL_INT *il , MKL_INT *iu , float *abstol , MKL_INT *m , MKL_INT *nz , float *w , float *orfac , MKL_Complex8 *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT *info );
void pzheevx (char *jobz , char *range , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc , double *vl , double *vu , MKL_INT *il , MKL_INT *iu , double *abstol , MKL_INT *m , MKL_INT *nz , double *w , double *orfac , MKL_Complex16 *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT *lrwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *ifail , MKL_INT *iclustr , double *gap , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?heevx routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A by calling the recommended sequence of ScaLAPACK routines. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

<i>jobz</i>	(global) CHARACTER*1. Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors: If <i>jobz</i> = 'N', then only eigenvalues are computed. If <i>jobz</i> = 'V', then eigenvalues and eigenvectors are computed.
<i>range</i>	(global) CHARACTER*1. Must be 'A', 'V', or 'I'. If <i>range</i> = 'A', all eigenvalues will be found. If <i>range</i> = 'V', all eigenvalues in the half-open interval [<i>vl</i> , <i>vu</i>] will be found. If <i>range</i> = 'I', the eigenvalues with indices <i>il</i> through <i>iu</i> will be found.
<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the Hermitian matrix <i>A</i> is stored: If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular part of <i>A</i> . If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular part of <i>A</i> .
<i>n</i>	(global) INTEGER. The number of rows and columns of the matrix <i>A</i> (<i>n</i> \geq 0).
<i>a</i>	(local). COMPLEX for pcheevx DOUBLE COMPLEX for pzheevx. Block cyclic array of global dimension (<i>n</i> , <i>n</i>) and local dimension (<i>lld_a</i> , LOC <i>c(ja+n-1)</i>). On entry, the Hermitian matrix <i>A</i> . If <i>uplo</i> = 'U', only the upper triangular part of <i>A</i> is used to define the elements of the Hermitian matrix. If <i>uplo</i> = 'L', only the lower triangular part of <i>A</i> is used to define the elements of the Hermitian matrix.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca(ctxt_)</i> is incorrect, p?heevx cannot guarantee correct error reporting.
<i>vl</i> , <i>vu</i>	(global) REAL for pcheevx DOUBLE PRECISION for pzheevx. If <i>range</i> = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; not referenced if <i>range</i> = 'A' or 'I'.
<i>il</i> , <i>iu</i>	(global)

INTEGER. If *range* = 'I', the indices of the smallest and largest eigenvalues to be returned.

Constraints:

il ≥ 1 ; $\min(il, n) \leq iu \leq n$.

Not referenced if *range* = 'A' or 'V'.

abstol

(global).

REAL for pcheevx

DOUBLE PRECISION for pzheevx.

If *jobz*='V', setting *abstol* to *p?lamch(context, 'U')* yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to *abstol*+*eps**max($|a|, |b|$), where *eps* is the machine precision. If *abstol* is less than or equal to zero, then *eps**norm(*T*) will be used in its place, where norm(*T*) is the 1-norm of the tridiagonal matrix obtained by reducing *A* to tridiagonal form.

Eigenvalues are computed most accurately when *abstol* is set to twice the underflow threshold $2*p?lamch('S')$, not zero. If this routine returns with ((mod(*info*, 2).ne.0).or.(mod(*info*/8, 2).ne.0)), indicating that some eigenvalues or eigenvectors did not converge, try setting *abstol* to $2*p?lamch('S')$.

orfac

(global). REAL for pcheevx

DOUBLE PRECISION for pzheevx.

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within *tol*=*orfac**norm(*A*) of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0e-3 is used if *orfac* is negative.

orfac should be identical on all processes.

iz, *jz*

(global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*. *descz(ctxt_)* must equal *desca(ctxt_)*.

work

(local).

COMPLEX for pcheevx

DOUBLE COMPLEX for pzheevx.

Array, size *lwork*.

lwork

(local) INTEGER. The dimension of the array *work*.

If only eigenvalues are requested:

lwork $\geq n + \max(nb * (np0 + 1), 3)$

If eigenvectors are requested:

lwork $\geq n + (np0 + mq0 + nb) * nb$

with $nq0 = \text{numroc}(nn, nb, 0, 0, \text{NPCOL})$.

lwork $\geq 5 * n + \max(5 * nn, np0 * mq0 + 2 * nb * nb) + \text{ceil}(neig, \text{NPROW} * \text{NPCOL}) * nn$

For optimal performance, greater workspace is needed, that is

lwork $\geq \max(lwork, nhetrd_lwork)$

where *lwork* is as defined above, and $nhetrd_lwork = n + 2 * (anb + 1) * (4 * nps + 2) + (nps + 1) * nps$

```

ictxt = desca(ctxt_)

anb = pjlaenv(ictxt, 3, 'pchetrd', 'L', 0, 0, 0, 0)
sqnpc = sqrt(dble(NPROW * NPCOL))
nps = max(numroc(n, 1, 0, 0, sqnpc), 2 * anb)

```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

rwork

(local)

REAL for pcheevx

DOUBLE PRECISION for pzheevx.

Workspace array, size *lrwork*.*lrwork*(local) INTEGER. The dimension of the array *work*.See below for definitions of variables used to define *lwork*.If no eigenvectors are requested (*jobz* = 'N'), then $lrwork \geq 5 * nn + 4 * n$.If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:
$$lrwork \geq 4 * n + \max(5 * nn, np0 * mq0 + 2 * nb * nb) + \text{ceil}(neig, \text{NPROW} * \text{NPCOL}) * nn$$
The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following values to *lrwork*:*(clustersize-1)*n*,where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:
$$\{w(k), \dots, w(k+clustersize-1) | w(j+1) \leq w(j) + orfac * 2 * \text{norm}(A)\}.$$

Variable definitions:

```

neig = number of eigenvectors requested;
nb = descra(mb_) = descra(nb_) = descz(mb_) = descz(nb_);
nn = max(n, NB, 2);
descra(rsra_) = descra(nb_) = descz(rsra_) = descz(csra_) = 0;
np0 = numroc(nn, nb, 0, 0, NPROW);
mq0 = numroc(max(neig, nb, 2), nb, 0, 0, NPCOL);
iceil(x, y) is a ScaLAPACK function returning ceiling(x/y)

```

When *lwork* is too small:

If *lwork* is too small to guarantee orthogonality, p?heevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues. If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned. Note that when *range*='V', p?heevx does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range*='V' and as long as *lwork* is large enough to allow p?heevx to compute the eigenvalues, p?heevx will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality and performance:

If *clustersize* $\geq n/\sqrt{NPROW \cdot NPCOL}$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, *clustersize* = *n*-1) p?stein will perform no better than ?stein on 1 processor.

For *clustersize* = *n*/ $\sqrt{NPROW \cdot NPCOL}$ reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For *clustersize* $> n/\sqrt{NPROW \cdot NPCOL}$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.

iwork

(local) INTEGER. Workspace array.

liwork

(local) INTEGER, dimension of *iwork*.

liwork $\geq 6 * nnp$

Where: *nnp* = max(*n*, NPROW*NPCOL+1, 4)

If *liwork* = -1, then *liwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

<i>a</i>	On exit, the lower triangle (if <i>uplo</i> = 'L'), or the upper triangle (if <i>uplo</i> = 'U') of <i>A</i> , including the diagonal, is overwritten.
<i>m</i>	(global) INTEGER. The total number of eigenvalues found; $0 \leq m \leq n$.
<i>nz</i>	(global) INTEGER. Total number of eigenvectors computed. $0 \leq nz \leq m$. The number of columns of <i>z</i> that are filled. If <i>jobz</i> ≠ 'V', <i>nz</i> is not referenced. If <i>jobz</i> = 'V', <i>nz</i> = <i>m</i> unless the user supplies insufficient space and p?heevx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in <i>z</i> (<i>m.le.descz(n_)</i>) and sufficient workspace to compute them. (See <i>lwork</i>). p?heevx is always able to detect insufficient space without computation unless <i>range.eq.</i> 'V'.
<i>w</i>	(global). REAL for pcheevx DOUBLE PRECISION for pzheevx. Array, size (<i>n</i>). The first <i>m</i> elements contain the selected eigenvalues in ascending order.
<i>z</i>	(local). COMPLEX for pcheevx DOUBLE COMPLEX for pzheevx. Array, global dimension (<i>n, n</i>), local dimension (<i>lld_z, LOCC(jz+n-1)</i>). If <i>jobz</i> = 'V', then on normal exit the first <i>m</i> columns of <i>z</i> contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of <i>z</i> contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in <i>ifail</i> . If <i>jobz</i> = 'N', then <i>z</i> is not referenced.
<i>work(1)</i>	On exit, returns adequate workspace to allow optimal performance.
<i>rwork</i>	(local). REAL for pcheevx DOUBLE PRECISION for pzheevx. Array, size (<i>lwork</i>). On return, <i>rwork(1)</i> contains the optimal amount of workspace required for efficient execution. If <i>jobz</i> = 'N' <i>rwork(1)</i> = optimal amount of workspace required to compute eigenvalues efficiently. If <i>jobz</i> = 'V' <i>rwork(1)</i> = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality. If <i>range</i> = 'V', it is assumed that all eigenvectors may be required.

iwork(1) (local)
 On return, *iwork*(1) contains the amount of integer workspace required.

ifail (global) INTEGER.
 Array, size (*n*).
 If *jobz* = 'V', then on normal exit, the first *m* elements of *ifail* are zero. If $(\text{mod}(\text{info}, 2) \neq 0)$ on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.
 If *jobz* = 'N', then *ifail* is not referenced.

iclustr (global) INTEGER.
 Array, size (2*NPROW*NPCOL).
 This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*(2*i-1) to *iclustr*(2*i), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr*() is a zero terminated array. (*iclustr*(2*k) .ne. 0. and. *iclustr*(2*k +1) .eq. 0) if and only if *k* is the number of clusters. *iclustr* is not referenced if *jobz* = 'N'.

gap (global)
 REAL for pcheevx
 DOUBLE PRECISION for pzheevx.
 Array, size (NPROW*NPCOL)
 This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the *i*-th cluster may be as high as $(C*n) / \text{gap}(i)$ where *C* is a small constant.

info (global) INTEGER.
 If *info* = 0, the execution is successful.
 If *info* < 0:
 If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*). If the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.
 If *info* > 0:
 If $(\text{mod}(\text{info}, 2) \neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in *ifail*. Ensure *abstol*=2.0*p?lamch('U')
 If $(\text{mod}(\text{info}/2, 2) \neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(info/4, 2) \neq 0)$, then space limit prevented `p?syevx` from computing all of the eigenvectors between v_l and v_u . The number of eigenvectors computed is returned in nz .

If $(\text{mod}(info/8, 2) \neq 0)$, then `p?stebz` failed to compute eigenvalues. Ensure $\text{abstol}=2.0 * \text{p?lamch}('U')$.

`p?gesvd`

Computes the singular value decomposition of a general matrix, optionally computing the left and/or right singular vectors.

Syntax

Fortran:

```
call psgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt,
descvt, work, lwork, info)

call pdgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt,
descvt, work, lwork, info)

call pcgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt,
descvt, work, lwork, rwork, info)

call pzgesvd(jobu, jobvt, m, n, a, ia, ja, desca, s, u, iu, ju, descu, vt, ivt, jvt,
descvt, work, lwork, rwork, info)
```

C:

```
void psgesvd (char *jobu , char *jobvt , MKL_INT *m , MKL_INT *n , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *s , float *u , MKL_INT *iu , MKL_INT *ju ,
MKL_INT *descu , float *vt , MKL_INT *ivt , MKL_INT *jvt , MKL_INT *descvt , float
*work , MKL_INT *lwork , float *rwork , MKL_INT *info );

void pdgesvd (char *jobu , char *jobvt , MKL_INT *m , MKL_INT *n , double *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , double *s , double *u , MKL_INT *iu , MKL_INT
*ju , MKL_INT *descu , double *vt , MKL_INT *ivt , MKL_INT *jvt , MKL_INT *descvt ,
double *work , MKL_INT *lwork , double *rwork , MKL_INT *info );

void pcgesvd (char *jobu , char *jobvt , MKL_INT *m , MKL_INT *n , MKL_Complex8 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *s , MKL_Complex8 *u , MKL_INT
*iu , MKL_INT *ju , MKL_INT *descu , MKL_Complex8 *vt , MKL_INT *ivt , MKL_INT *jvt ,
MKL_INT *descvt , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT *info );

void pzgesvd (char *jobu , char *jobvt , MKL_INT *m , MKL_INT *n , MKL_Complex16 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *s , MKL_Complex16 *u , MKL_INT
*iu , MKL_INT *ju , MKL_INT *descu , MKL_Complex16 *vt , MKL_INT *ivt , MKL_INT *jvt ,
MKL_INT *descvt , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT
*info );
```

Include Files

- C: `mkl_scalapack.h`

Description

The `p?gesvd` routine computes the singular value decomposition (SVD) of an m -by- n matrix A , optionally computing the left and/or right singular vectors. The SVD is written

$$A = U \Sigma V^T,$$

where Σ is an m -by- n matrix that is zero except for its $\min(m, n)$ diagonal elements, U is an m -by- m orthogonal matrix, and V is an n -by- n orthogonal matrix. The diagonal elements of Σ are the singular values of A and the columns of U and V are the corresponding right and left singular vectors, respectively. The singular values are returned in array s in decreasing order and only the first $\min(m, n)$ columns of U and rows of $vt = V^T$ are computed.

Input Parameters

mp = number of local rows in A and U

nq = number of local columns in A and VT

$size = \min(m, n)$

$sizeq$ = number of local columns in U

$sizep$ = number of local rows in VT

$jobu$ (global) CHARACTER*1. Specifies options for computing all or part of the matrix U .

If $jobu = 'V'$, the first $size$ columns of U (the left singular vectors) are returned in the array u ;

If $jobu = 'N'$, no columns of U (no left singular vectors) are computed.

$jobvt$

(global) CHARACTER*1.

Specifies options for computing all or part of the matrix V^T .

If $jobvt = 'V'$, the first $size$ rows of V^T (the right singular vectors) are returned in the array vt ;

If $jobvt = 'N'$, no rows of V^T (no right singular vectors) are computed.

m

(global) INTEGER. The number of rows of the matrix A ($m \geq 0$).

n

(global) INTEGER. The number of columns in A ($n \geq 0$).

a

(local). REAL for psgesvd

DOUBLE PRECISION for pdgesvd

COMPLEX for pcgesvd

COMPLEX*16 for pzgesvd

Block cyclic array, global dimension (m, n) , local dimension (mp, nq) .

$work(lwork)$ is a workspace array.

ia, ja

(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

$desca$

(global and local) INTEGER array, dimension $(dlen_)$. The array descriptor for the distributed matrix A .

iu, ju

(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix U , respectively.

<i>descu</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>U</i> .
<i>ivt, jvt</i>	(global) INTEGER. The row and column indices in the global array <i>vt</i> indicating the first row and the first column of the submatrix <i>VT</i> , respectively.
<i>descvt</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>VT</i> .
<i>work</i>	(local). REAL for psgesvd DOUBLE PRECISION for pdgesvd COMPLEX for pcgesvd COMPLEX*16 for pzgesvd Workspace array, dimension (<i>lwork</i>)
<i>lwork</i>	(local) INTEGER. The dimension of the array <i>work</i> ; $lwork > 2 + 6 * sizeb + \max(watobd, wbdtosvd),$ where <i>sizeb</i> = $\max(m, n)$, and <i>watobd</i> and <i>wbdtosvd</i> refer, respectively, to the workspace required to bidiagonalize the matrix <i>A</i> and to go from the bidiagonal matrix to the singular value decomposition <i>U S VT</i> . For <i>watobd</i> , the following holds: $watobd = \max(\max(wp?lange, wp?gebrd), \max(wp?lared2d, wp?lared1d)),$ where <i>wp?lange</i> , <i>wp?lared1d</i> , <i>wp?lared2d</i> , <i>wp?gebrd</i> are the workspaces required respectively for the subprograms <i>p?lange</i> , <i>p?lared1d</i> , <i>p?lared2d</i> , <i>p?gebrd</i> . Using the standard notation $mp = \text{numroc}(m, mb, \text{MYROW}, \text{desca}(\text{ctxt}_{}), \text{NPROW}),$ $nq = \text{numroc}(n, nb, \text{MYCOL}, \text{desca}(lld_), \text{NPCOL}),$ the workspaces required for the above subprograms are $wp?lange = mp,$ $wp?lared1d = nq,$ $wp?lared2d = mp,$ $wp?gebrd = nb * (mp + nq + 1) + nq,$ where <i>nq</i> and <i>mp</i> refer, respectively, to the values obtained at <i>MYCOL</i> = 0 and <i>MYROW</i> = 0. In general, the upper limit for the workspace is given by a workspace required on processor (0,0): $watobd \leq nb * (mp + nq + 1) + nq.$ In case of a homogeneous process grid this upper limit can be used as an estimate of the minimum workspace for every processor. For <i>wbdtosvd</i> , the following holds: $wbdtosvd = size * (wantu * nru + wantvt * ncvt) + \max(w?bdsqr, \max(wantu * wp?ormbrqln, wantvt * wp?ormbrprt)),$ where

`wantu(wantvt) = 1`, if left/right singular vectors are wanted, and
`wantu(wantvt) = 0`, otherwise. `w?bdsqr`, `wp?ormbrqln`, and `wp?ormbrprt`
refer respectively to the workspace required for the subprograms `?bdsqr`,
`p?ormbr(qln)`, and `p?ormbr(prt)`, where `qln` and `prt` are the values of the
arguments `vect`, `side`, and `trans` in the call to `p?ormbr`. `nru` is equal to the
local number of rows of the matrix U when distributed 1-dimensional
"column" of processes. Analogously, `ncvt` is equal to the local number of
columns of the matrix VT when distributed across 1-dimensional "row" of
processes. Calling the LAPACK procedure `?bdsqr` requires

`w?bdsqr = max(1, 2*size + (2*size - 4)* max(wantu, wantvt))`
on every processor. Finally,

`wp?ormbrqln = max((nb*(nb-1))/2, (sizeq+mp)*nb)+nb*nb,`

`wp?ormbrprt = max((mb*(mb-1))/2, (sizep+nq)*mb)+mb*mb,`

If `lwork = -1`, then `lwork` is global input and a workspace query is
assumed; the routine only calculates the minimum size for the work array.
The required workspace is returned as the first element of `work` and no
error message is issued by `xerbla`.

<code>rwork</code>	<code>REAL for pcgesvd</code> <code>DOUBLE PRECISION for pzgesvd</code> Workspace array of size $1 + 4*sizeb$. Not used for <code>psgesvd</code> and <code>pdgesvd</code> .
--------------------	--

Output Parameters

<code>a</code>	On exit, the contents of <code>a</code> are destroyed.
<code>s</code>	(global). <code>REAL for psgesvd</code> and <code>pcgesvd</code> <code>DOUBLE PRECISION for pdgesvd</code> and <code>pzgesvd</code> Array of size <code>size</code> . Contains the singular values of A sorted so that $s(i) \geq s(i+1)$.
<code>u</code>	(local). <code>REAL for psgesvd</code> <code>DOUBLE PRECISION for pdgesvd</code> <code>COMPLEX for pcgesvd</code> <code>COMPLEX*16 for pzgesvd</code> local dimension (<code>mp, sizeq</code>), global dimension (<code>m, size</code>) If <code>jobu = 'V'</code> , <code>u</code> contains the first $\min(m, n)$ columns of U . If <code>jobu = 'N'</code> or <code>'O'</code> , <code>u</code> is not referenced.
<code>vt</code>	(local). <code>REAL for psgesvd</code> <code>DOUBLE PRECISION for pdgesvd</code> <code>COMPLEX for pcgesvd</code> <code>COMPLEX*16 for pzgesvd</code> local dimension (<code>sizep, nq</code>), global dimension (<code>size, n</code>)

If $\text{jobvt} = \text{'V'}$, vt contains the first size rows of V^T if $\text{jobu} = \text{'N'}$, vt is not referenced.

work On exit, if $\text{info} = 0$, then $\text{work}(1)$ returns the required minimal size of lwork .

rwork On exit, if $\text{info} = 0$, then $\text{rwork}(1)$ returns the required size of rwork .

info (global) INTEGER.

If $\text{info} = 0$, the execution is successful.

If $\text{info} < 0$, If $\text{info} = -i$, the i th parameter had an illegal value.

If $\text{info} > 0$ i , then if ?bdsqr did not converge,

If $\text{info} = \min(m, n) + 1$, then p?gesvd has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from p?gesvd cannot be guaranteed.

See Also

?bdsqr
p?ormbr
pxerbla

p?sygvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

Fortran:

```
call pssygvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb, vl,
vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork,
ifail, iclustr, gap, info)
```

```
call pdsygvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb, vl,
vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, iwork, liwork,
ifail, iclustr, gap, info)
```

C:

```
void pssygvx (MKL_INT *ibtype , char *jobz , char *range , char *uplo , MKL_INT *n ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , float *vl , float *vu , MKL_INT *il , MKL_INT *iu ,
float *abstol , MKL_INT *m , MKL_INT *nz , float *w , float *orfac , float *z ,
MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , float *work , MKL_INT *lwork , MKL_INT
*iwork , MKL_INT *liwork , MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT
*info );
```

```
void pdsygvx (MKL_INT *ibtype , char *jobz , char *range , char *uplo , MKL_INT *n ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , double *vl , double *vu , MKL_INT *il , MKL_INT *iu ,
double *abstol , MKL_INT *m , MKL_INT *nz , double *w , double *orfac , double *z ,
MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , double *work , MKL_INT *lwork , MKL_INT
*iwork , MKL_INT *liwork , MKL_INT *ifail , MKL_INT *iclustr , double *gap , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?sygsvx` routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$\text{sub}(A) * \mathbf{x} = \lambda * \text{sub}(B) * \mathbf{x}, \text{sub}(A)^\top * \mathbf{x} = \lambda * \mathbf{x}, \text{or } \text{sub}(B) * \text{sub}(A) * \mathbf{x} = \lambda * \mathbf{x}.$$

Here \mathbf{x} denotes eigen vectors, λ (*lambda*) denotes eigenvalues, $\text{sub}(A)$ denoting $A_{(ia:ia+n-1, ja:ja+n-1)}$ is assumed to symmetric, and $\text{sub}(B)$ denoting $B_{(ib:ib+n-1, jb:jb+n-1)}$ is also positive definite.

Input Parameters

<code>ibtype</code>	(global) INTEGER. Must be 1 or 2 or 3. Specifies the problem type to be solved: If <code>ibtype</code> = 1, the problem type is $\text{sub}(A) * \mathbf{x} = \lambda * \text{sub}(B) * \mathbf{x}$; If <code>ibtype</code> = 2, the problem type is $\text{sub}(A)^\top * \mathbf{x} = \lambda * \mathbf{x}$; If <code>ibtype</code> = 3, the problem type is $\text{sub}(B) * \text{sub}(A) * \mathbf{x} = \lambda * \mathbf{x}$.
<code>jobz</code>	(global) CHARACTER*1. Must be ' <code>N</code> ' or ' <code>V</code> '. If <code>jobz</code> = ' <code>N</code> ', then compute eigenvalues only. If <code>jobz</code> = ' <code>V</code> ', then compute eigenvalues and eigenvectors.
<code>range</code>	(global) CHARACTER*1. Must be ' <code>A</code> ' or ' <code>V</code> ' or ' <code>I</code> '. If <code>range</code> = ' <code>A</code> ', the routine computes all eigenvalues. If <code>range</code> = ' <code>V</code> ', the routine computes eigenvalues in the interval: [<code>vl</code> , <code>vu</code>] If <code>range</code> = ' <code>I</code> ', the routine computes eigenvalues with indices <code>il</code> through <code>iu</code> .
<code>uplo</code>	(global) CHARACTER*1. Must be ' <code>U</code> ' or ' <code>L</code> '. If <code>uplo</code> = ' <code>U</code> ', arrays <code>a</code> and <code>b</code> store the upper triangles of $\text{sub}(A)$ and $\text{sub}(B)$; If <code>uplo</code> = ' <code>L</code> ', arrays <code>a</code> and <code>b</code> store the lower triangles of $\text{sub}(A)$ and $\text{sub}(B)$.
<code>n</code>	(global) INTEGER. The order of the matrices $\text{sub}(A)$ and $\text{sub}(B)$, $n \geq 0$.
<code>a</code>	(local) REAL for <code>pssygvx</code> DOUBLE PRECISION for <code>pdsygvx</code> . Pointer into the local memory to an array of dimension (<code>lld_a</code> , <code>LOCc(ja+n-1)</code>). On entry, this array contains the local pieces of the n -by- n symmetric distributed matrix $\text{sub}(A)$. If <code>uplo</code> = ' <code>U</code> ', the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix.

If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix.

ia, ja

(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

desca

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix A . If $desca(ctxt_)$ is incorrect, $p?sygsvx$ cannot guarantee correct error reporting.

b

(local). REAL for $pssygvx$

DOUBLE PRECISION for $pdsygvx$.

Pointer into the local memory to an array of dimension (lld_b , $LOCc(jb + n - 1)$). On entry, this array contains the local pieces of the n -by- n symmetric distributed matrix $\text{sub}(B)$.

If $uplo = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(B)$ contains the upper triangular part of the matrix.

If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix.

ib, jb

(global) INTEGER. The row and column indices in the global array b indicating the first row and the first column of the submatrix B , respectively.

descb

(global and local) INTEGER array, dimension ($dlen_$). The array descriptor for the distributed matrix B . $descb(ctxt_)$ must be equal to $desca(ctxt_)$.

vl, vu

(global)

REAL for $pssygvx$

DOUBLE PRECISION for $pdsygvx$.

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.

If $range = 'A'$ or ' I' , vl and vu are not referenced.

il, iu

(global)

INTEGER.

If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $il \geq 1$, $\min(il, n) \leq iu \leq n$

If $range = 'A'$ or ' V' , il and iu are not referenced.

abstol

(global)

REAL for $pssygvx$

DOUBLE PRECISION for $pdsygvx$.

If $jobz='V'$, setting $abstol$ to $p?lamch(context, 'U')$ yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to

$$abstol + \text{eps} * \max(|a|, |b|),$$

where eps is the machine precision. If $abstol$ is less than or equal to zero, then $\text{eps} * \text{norm}(T)$ will be used in its place, where $\text{norm}(T)$ is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * \text{p?lamch('S')}$ not zero. If this routine returns with $((\text{mod}(\text{info}, 2) .ne. 0) .or. * (\text{mod}(\text{info}/8, 2) .ne. 0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting $abstol$ to $2 * \text{p?lamch('S')}$.

orfac

(global).

REAL for pssygvx

DOUBLE PRECISION for pdsvyvx.

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol = orfac * \text{norm}(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see *lwork*), *tol* may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of $1.0e-3$ is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, jz

(global) INTEGER. The row and column indices in the global array Z indicating the first row and the first column of the submatrix Z , respectively.

descz

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix Z . *descz(ctxt_)* must equal *desca(ctxt_)*.

work

(local)

REAL for pssygvx

DOUBLE PRECISION for pdsvyvx.

Workspace array, dimension (*lwork*)

lwork

(local) INTEGER.

Dimension of the array *work*. See below for definitions of variables used to define *lwork*.

If no eigenvectors are requested (*jobz* = 'N'), then $lwork \geq 5 * n + \max(5 * nn, NB * (np0 + 1))$.

If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

$$lwork \geq 5 * n + \max(5 * nn, np0 * mq0 + 2 * nb * nb) + \text{ceil}(neig, \text{NPROW} * \text{NPCOL}) * nn.$$

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality at the cost of potentially poor performance you should add the following to *lwork*:

*(clustersize-1)*n,*

where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

```
{ w(k) , . . . , w(k+clustersize-1) | w(j+1) ≤ w(j) +  
orfac*2*norm(A) }
```

Variable definitions:

neig = number of eigenvectors requested,

nb = *desca(mb_)* = *desca(nb_)* = *descz(mb_)* = *descz(nb_)*,

nn = *max(n, nb, 2),*

desca(rsrcc_) = *desca(nb_)* = *descz(rsrcc_)* = *descz(csrrc_)* = 0,

np0 = *numroc(nn, nb, 0, 0, NPROW),*

mq0 = *numroc(max(neig, nb, 2), nb, 0, 0, NPCOL)*

ceil(x, y) is a ScaLAPACK function returning ceiling(*x/y*)

If *lwork* is too small to guarantee orthogonality, p?syevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -23 is returned.

Note that when *range*='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if *lwork* is large enough to compute the eigenvalues, p?sygvx computes the eigenvalues and as many eigenvectors as possible.

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

lwork ≥ max(*lwork*, 5*n + *nsytrd_lwopt*, *nsygst_lwopt*), where

lwork, as defined previously, depends upon the number of eigenvectors requested, and

nsytrd_lwopt = *n* + 2*(*anb*+1)*(4*nps+2) + (*nps*+3)**nps*

nsygst_lwopt = 2**np0***nb* + *nq0***nb* + *nb***nb*

anb = *pjlaenv(desca(ctxt_), 3, p?sytrd ', 'L', 0, 0, 0, 0)*

sqrnpc = int(sqrt(db1e(NPROW * NPCOL)))

nps = max(numroc(*n*, 1, 0, 0, *sqrnpc*), 2**anb*)

NB = *desca(mb_)*

np0 = *numroc(n, nb, 0, 0, NPROW)*

nq0 = *numroc(n, nb, 0, 0, NPCOL)*

numroc is a ScaLAPACK tool functions;

pjlaenv is a ScaLAPACK environmental inquiry function

MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine `blacs_gridinfo`.

For large n , no extra workspace is needed, however the biggest boost in performance comes for small n , so it is wise to provide the extra workspace (typically less than a Megabyte per process).

If $clustersize \geq n/\sqrt{NPROW \cdot NPCOL}$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, $clustersize = n-1$) `p?stein` will perform no better than `?stein` on a single processor.

For $clustersize = n/\sqrt{NPROW \cdot NPCOL}$ reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For $clustersize > n/\sqrt{NPROW \cdot NPCOL}$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pxerbla`.

$iwork$

(local) INTEGER. Workspace array.

$liwork$

(local) INTEGER, dimension of $iwork$.

$liwork \geq 6 * nnp$

Where:

$nnp = \max(n, NPROW \cdot NPCOL + 1, 4)$

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

a

On exit,

If $jobz = 'V'$, and if $info = 0$, `sub(A)` contains the distributed matrix Z of eigenvectors. The eigenvectors are normalized as follows:

```
for ibtype = 1 or 2,  $Z^T * sub(B) * Z = i$ ;
for ibtype = 3,  $Z^T * inv(sub(B)) * Z = i$ .
```

If $jobz = 'N'$, then on exit the upper triangle (if $uplo='U'$) or the lower triangle (if $uplo='L'$) of `sub(A)`, including the diagonal, is destroyed.

b

On exit, if $info \leq n$, the part of `sub(B)` containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $\text{sub}(B) = U^T * U$ or $\text{sub}(B) = L * L^T$.

m

(global) INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.

nz

(global) INTEGER.

Total number of eigenvectors computed. $0 \leq nz \leq m$. The number of columns of *z* that are filled.

If *jobz* ≠ 'V', *nz* is not referenced.

If *jobz* = 'V', *nz* = *m* unless the user supplies insufficient space and p?sygvx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in *z* (m.le.descz(*n_*)) and sufficient workspace to compute them. (See *lwork* below.) p?sygvx is always able to detect insufficient space without computation unless *range.eq.*'V'.

w

(global)

REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

Array, size (*n*). On normal exit, the first *m* entries contain the selected eigenvalues in ascending order.

z

(local).

REAL for pssygvx

DOUBLE PRECISION for pdsygvx.

global dimension (*n*, *n*), local dimension (lld_z, locc(jz+n-1)).

If *jobz* = 'V', then on normal exit the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of *z* contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in *ifail*.

If *jobz* = 'N', then *z* is not referenced.

work

If *jobz*='N' *work*(1) = optimal amount of workspace required to compute eigenvalues efficiently

If *jobz* = 'V' *work*(1) = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If *range*='V', it is assumed that all eigenvectors may be required.

ifail

(global) INTEGER.

Array, size (*n*).

ifail provides additional information when *info.ne.0*

If (mod(*info*/16, 2).ne.0) then *ifail*(1) indicates the order of the smallest minor which is not positive definite. If (mod(*info*, 2).ne.0) on exit, then *ifail* contains the indices of the eigenvectors that failed to converge.

If neither of the above error conditions hold and *jobz* = 'V', then the first *m* elements of *ifail* are set to zero.

iclustr

(global) INTEGER.

Array, size $(2 * \text{NPROW} * \text{NPCOL})$. This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*($2*i-1$) to *iclustr*($2*i$), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. *iclustr()* is a zero terminated array.

$(\text{iclustr}(2*k) .ne. 0 .and. \text{iclustr}(2*k+1) .eq. 0)$ if and only if k is the number of clusters *iclustr* is not referenced if *jobz* = 'N'.

gap

(global)

REAL for pssygvx

DOUBLE PRECISION for pdsgvxx.

Array, size $(\text{NPROW} * \text{NPCOL})$. This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the i -th cluster may be as high as $(C*n) / \text{gap}(i)$, where C is a small constant.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0: the i -th argument is an array and the j -entry had an illegal value, then *info* = $-(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then *info* = $-i$.

If *info* > 0:

If $(\text{mod}(\text{info}, 2) .ne. 0)$, then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.

If $(\text{mod}(\text{info}, 2, 2) .ne. 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(\text{info}/4, 2) .ne. 0)$, then space limit prevented p?sygvx from computing all of the eigenvectors between *vl* and *. The number of eigenvectors computed is returned in *nz*.*

If $(\text{mod}(\text{info}/8, 2) .ne. 0)$, then p?stebz failed to compute eigenvalues.

If $(\text{mod}(\text{info}/16, 2) .ne. 0)$, then *B* was not positive definite. *ifail*(1) indicates the order of the smallest minor which is not positive definite.

p?hegvx

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax

Fortran:

```
call pchegvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb, vl,
vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork,
iwork, liwork, ifail, iclustr, gap, info)
```

```
call pzhegvx(ibtype, jobz, range, uplo, n, a, ia, ja, desca, b, ib, jb, descb, vl,
vu, il, iu, abstol, m, nz, w, orfac, z, iz, jz, descz, work, lwork, rwork, lrwork,
iwork, liwork, ifail, iclustr, gap, info)
```

C:

```
void pchegvx (MKL_INT *ibtype , char *jobz , char *range , char *uplo , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *vl , float *vu , MKL_INT *il ,
MKL_INT *iu , float *abstol , MKL_INT *m , MKL_INT *nz , float *w , float *orfac ,
MKL_Complex8 *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , MKL_Complex8 *work ,
MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *ifail , MKL_INT *iclustr , float *gap , MKL_INT *info );

void pzhegvx (MKL_INT *ibtype , char *jobz , char *range , char *uplo , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *vl , double *vu , MKL_INT *il ,
MKL_INT *iu , double *abstol , MKL_INT *m , MKL_INT *nz , double *w , double *orfac ,
MKL_Complex16 *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , MKL_Complex16 *work ,
MKL_INT *lwork , double *rwork , MKL_INT *lrwork , MKL_INT *iwork , MKL_INT *liwork ,
MKL_INT *ifail , MKL_INT *iclustr , double *gap , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?hegvx routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$$\text{sub}(A) \cdot x = \lambda \cdot \text{sub}(B) \cdot x, \quad \text{sub}(A) \cdot \text{sub}(B) \cdot x = \lambda \cdot x, \quad \text{or} \quad \text{sub}(B) \cdot \text{sub}(A) \cdot x = \lambda \cdot x.$$

Here $\text{sub}(A)$ denoting $A(ia:ia+n-1, ja:ja+n-1)$ and $\text{sub}(B)$ are assumed to be Hermitian and $\text{sub}(B)$ denoting $B(ib:ib+n-1, jb:jb+n-1)$ is also positive definite.

Input Parameters

ibtype (global) INTEGER. Must be 1 or 2 or 3.

Specifies the problem type to be solved:

If *ibtype* = 1, the problem type is

$$\text{sub}(A) \cdot x = \lambda \cdot \text{sub}(B) \cdot x;$$

If *ibtype* = 2, the problem type is

$$\text{sub}(A) \cdot \text{sub}(B) \cdot x = \lambda \cdot x;$$

If *ibtype* = 3, the problem type is

$$\text{sub}(B) \cdot \text{sub}(A) \cdot x = \lambda \cdot x.$$

jobz (global) CHARACTER*1. Must be 'N' or 'V'.

If *jobz* ='N', then compute eigenvalues only.

If *jobz* ='V', then compute eigenvalues and eigenvectors.

<i>range</i>	(global) CHARACTER*1. Must be 'A' or 'V' or 'I'. If <i>range</i> = 'A', the routine computes all eigenvalues. If <i>range</i> = 'V', the routine computes eigenvalues in the interval: [<i>vl</i> , <i i="" vu<="">] If <i>range</i> = 'I', the routine computes eigenvalues with indices <i>il</i> through <i>iu</i>.</i>
<i>uplo</i>	(global) CHARACTER*1. Must be 'U' or 'L'. If <i>uplo</i> = 'U', arrays <i>a</i> and <i>b</i> store the upper triangles of sub(<i>A</i>) and sub(<i>B</i>); If <i>uplo</i> = 'L', arrays <i>a</i> and <i>b</i> store the lower triangles of sub(<i>A</i>) and sub(<i>B</i>).
<i>n</i>	(global) INTEGER. The order of the matrices sub(<i>A</i>) and sub (<i>B</i>) (<i>n</i> ≥ 0).
<i>a</i>	(local) COMPLEX for pchegvx DOUBLE COMPLEX for pzhegvx. Pointer into the local memory to an array of dimension (<i>lld_a</i> , <i>LOCc(ja</i> + <i>n</i> -1)). On entry, this array contains the local pieces of the <i>n</i> -by- <i>n</i> Hermitian distributed matrix sub(<i>A</i>). If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of sub(<i>A</i>) contains the upper triangular part of the matrix. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of sub(<i>A</i>) contains the lower triangular part of the matrix.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, dimension (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> . If <i>desca(ctxt_)</i> is incorrect, p?hegsv cannot guarantee correct error reporting.
<i>b</i>	(local). COMPLEX for pchegvx DOUBLE COMPLEX for pzhegvx. Pointer into the local memory to an array of dimension (<i>lld_b</i> , <i>LOCc(jb</i> + <i>n</i> -1)). On entry, this array contains the local pieces of the <i>n</i> -by- <i>n</i> Hermitian distributed matrix sub(<i>B</i>). If <i>uplo</i> = 'U', the leading <i>n</i> -by- <i>n</i> upper triangular part of sub(<i>B</i>) contains the upper triangular part of the matrix. If <i>uplo</i> = 'L', the leading <i>n</i> -by- <i>n</i> lower triangular part of sub(<i>B</i>) contains the lower triangular part of the matrix.
<i>ib, jb</i>	(global) INTEGER.

The row and column indices in the global array b indicating the first row and the first column of the submatrix B , respectively.

descb

(global and local) INTEGER array, dimension ($dlen_$).

The array descriptor for the distributed matrix B . $descb(ctxt_)$ must be equal to $desca(ctxt_)$.

vl, vu

(global)

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.

If $range = 'A'$ or $'I'$, vl and vu are not referenced.

il, iu

(global)

INTEGER.

If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $il \geq 1$, $\min(il, n) \leq iu \leq n$

If $range = 'A'$ or $'V'$, il and iu are not referenced.

abstol

(global)

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

If $jobz='V'$, setting $abstol$ to $p?lamch(context, 'U')$ yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to

$abstol + eps * \max(|a|, |b|)$,

where eps is the machine precision. If $abstol$ is less than or equal to zero, then $eps * \text{norm}(T)$ will be used in its place, where $\text{norm}(T)$ is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form.

Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2 * p?lamch('S')$ not zero. If this routine returns with $((\text{mod}(info, 2) .ne. 0) .or. * (\text{mod}(info/8, 2) .ne. 0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting $abstol$ to $2 * p?lamch('S')$.

orfac

(global).

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol = orfac * \text{norm}(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see $lwork$), tol may be decreased until all eigenvectors to be

reorthogonalized can be stored in one process. No reorthogonalization will be done if *orfac* equals zero. A default value of 1.0E-3 is used if *orfac* is negative. *orfac* should be identical on all processes.

iz, jz

(global) INTEGER. The row and column indices in the global array *z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz

(global and local) INTEGER array, dimension (*dlen_*). The array descriptor for the distributed matrix *Z*. *descz(ctxt_)* must equal *desca(ctxt_)*.

work

(local)

COMPLEX for *pchegvx*

DOUBLE COMPLEX for *pzhegvx*.

Workspace array, dimension (*lwork*)

lwork

(local).

INTEGER. The dimension of the array *work*.

If only eigenvalues are requested:

lwork $\geq n + \max(NB * (np0 + 1), 3)$

If eigenvectors are requested:

lwork $\geq n + (np0 + mq0 + NB) * NB$

with *mq0* = *numroc(nn, NB, 0, 0, NPCOL)*.

For optimal performance, greater workspace is needed, that is

lwork $\geq \max(lwork, n, nhetrdr_lwopt, nhegst_lwopt)$

where *lwork* is as defined above, and

nhetrdr_lwork = $2 * (anb + 1) * (4 * nps + 2) + (nps + 1) * nps;$

nhegst_lwopt = $2 * np0 * nb + mq0 * nb + nb * nb$

nb = *desca(mb_)*

np0 = *numroc(n, nb, 0, 0, NPROW)*

mq0 = *numroc(n, nb, 0, 0, NPCOL)*

ictxt = *desca(ctxt_)*

anb = *pjlaenv(ictxt, 3, 'p?hetrd', 'L', 0, 0, 0, 0)*

sqrnpc = *sqrt(dble(NPROW * NPCOL))*

nps = *max(numroc(n, 1, 0, 0, sqnpc), 2 * anb)*

numroc is a ScaLAPACK tool functions;

pjlaenv is a ScaLAPACK environmental inquiry function *MYROW*, *MYCOL*, *NPROW* and *NPCOL* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pixerbla.

rwork

(local)

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

Workspace array, size (*lrwork*).*lrwork*(local) INTEGER. The dimension of the array *rwork*.See below for definitions of variables used to define *lrwork*.If no eigenvectors are requested (*jobz* = 'N'), then *lrwork* $\geq 5*nn+4*n$ If eigenvectors are requested (*jobz* = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:*lrwork* $\geq 4*n + \max(5*nn, np0*mq0) + \text{ceil}(neig, \text{NPROW}*\text{NPCOL}) * nn$ The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and *orfac* is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following value to *lrwork*:*(clustersize-1)*n*,where *clustersize* is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:{*w(k), ..., w(k+clustersize-1)* | *w(j+1) ≤ w(j)+orfac*2*norm(A)*}

Variable definitions:

neig = number of eigenvectors requested;*nb* = *desca(mb_)* = *desca(nb_)* = *descz(mb_)* = *descz(nb_)*;*nn* = *max(n, nb, 2)*;*desca(rsra_)* = *desca(nb_)* = *descz(rsra_)* = *descz(csra_)* = 0 ;*np0* = *numroc(nn, nb, 0, 0, NPROW)*;*mq0* = *numroc(max(neig, nb, 2), nb, 0, 0, NPCOL)*;*ceil(x, y)* is a ScaLAPACK function returning ceiling(*x/y*).When *lwork* is too small:If *lwork* is too small to guarantee orthogonality, p?hegvx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.If *lwork* is too small to compute all the eigenvectors requested, no computation is performed and *info*= -25 is returned. Note that when *range='V'*, p?hegvx does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when *range='V'* and as long as *lwork* is large enough to allow p?hegvx to compute the eigenvalues, p?hegvx will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality & performance:

If $\text{clustersize} > n/\sqrt{(\text{NPROW} \times \text{NPCOL})}$, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, $\text{clustersize} = n-1$) `p?stein` will perform no better than `?stein` on 1 processor.

For $\text{clustersize} = n/\sqrt{(\text{NPROW} \times \text{NPCOL})}$ reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For $\text{clustersize} > n/\sqrt{(\text{NPROW} \times \text{NPCOL})}$ execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If $\text{lwork} = -1$, then liwork is global input and a workspace query is assumed; the routine only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by `pixerbla`.

$iwork$

(local) INTEGER. Workspace array.

$liwork$

(local) INTEGER, dimension of $iwork$.

$liwork \geq 6 * nnp$

Where: $nnp = \max(n, \text{NPROW} * \text{NPCOL} + 1, 4)$

If $liwork = -1$, then $liwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pixerbla`.

Output Parameters

a

On exit, if $\text{jobz} = 'V'$, then if $\text{info} = 0$, `sub(A)` contains the distributed matrix Z of eigenvectors.

The eigenvectors are normalized as follows:

If $\text{ibtype} = 1$ or 2 , then $Z^H * \text{sub}(B) * Z = i$;

If $\text{ibtype} = 3$, then $Z^H * \text{inv}(\text{sub}(B)) * Z = i$.

If $\text{jobz} = 'N'$, then on exit the upper triangle (if $\text{uplo}='U'$) or the lower triangle (if $\text{uplo}='L'$) of `sub(A)`, including the diagonal, is destroyed.

b

On exit, if $\text{info} \leq n$, the part of `sub(B)` containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization $\text{sub}(B) = U^H * U$, or $\text{sub}(B) = L * L^H$.

m

(global) INTEGER. The total number of eigenvalues found, $0 \leq m \leq n$.

nz

(global) INTEGER. Total number of eigenvectors computed. $0 < nz < m$. The number of columns of z that are filled.

If $\text{jobz} \neq 'V'$, nz is not referenced.

If $jobz = 'V'$, $nz = m$ unless the user supplies insufficient space and p?hegvx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in z (m. i.e. $descz(n_)$) and sufficient workspace to compute them. (See $lwork$ below.) The routine p?hegvx is always able to detect insufficient space without computation unless $range = 'V'$.

w

(global)

REAL for pchegvx

DOUBLE PRECISION for pzhegvx.

Array, size (n). On normal exit, the first m entries contain the selected eigenvalues in ascending order.*z*

(local).

COMPLEX for pchegvx

DOUBLE COMPLEX for pzhegvx.

global dimension (n, n), local dimension ($lld_z, locc(jz+n-1)$).If $jobz = 'V'$, then on normal exit the first m columns of z contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in $ifail$.If $jobz = 'N'$, then z is not referenced.*work*On exit, $work(1)$ returns the optimal amount of workspace.*rwork*On exit, $rwork(1)$ contains the amount of workspace required for optimal efficiencyIf $jobz='N'$ $rwork(1) =$ optimal amount of workspace required to compute eigenvalues efficientlyIf $jobz='V'$ $rwork(1) =$ optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.If $range='V'$, it is assumed that all eigenvectors may be required when computing optimal workspace.*ifail*

(global) INTEGER.

Array, size (n).*ifail* provides additional information when $info.ne.0$ If $(mod(info/16,2).ne.0)$, then $ifail(1)$ indicates the order of the smallest minor which is not positive definite.If $(mod(info,2).ne.0)$ on exit, then $ifail(1)$ contains the indices of the eigenvectors that failed to converge.If neither of the above error conditions are held, and $jobz = 'V'$, then the first m elements of $ifail$ are set to zero.*iclustr*

(global) INTEGER.

Array, size $(2 * \text{NPROW} * \text{NPCOL})$. This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see *lwork*, *orfac* and *info*). Eigenvectors corresponding to clusters of eigenvalues indexed *iclustr*($2*i-1$) to *iclustr*($2*i$), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal.

iclustr() is a zero terminated array. (*iclustr*($2*k$).ne.

$0.\text{and}. \text{clustr}(2*k+1).\text{eq.}0$) if and only if k is the number of clusters.

iclustr is not referenced if *jobz* = 'N'.

gap

(global)

REAL for *pchegvx*

DOUBLE PRECISION for *pzhegvx*.

Array, size $(\text{NPROW} * \text{NPCOL})$.

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array *iclustr*. As a result, the dot product between eigenvectors corresponding to the i -th cluster may be as high as $(C*n) / \text{gap}(i)$, where C is a small constant.

info

(global) INTEGER.

If *info* = 0, the execution is successful.

If *info* < 0: the i -th argument is an array and the j -entry had an illegal value, then *info* = $-(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then *info* = $-i$.

If *info* > 0:

If $(\text{mod}(\text{info}, 2).\text{ne.}0)$, then one or more eigenvectors failed to converge. Their indices are stored in *ifail*.

If $(\text{mod}(\text{info}, 2, 2).\text{ne.}0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array *iclustr*.

If $(\text{mod}(\text{info}/4, 2).\text{ne.}0)$, then space limit prevented *p?sygvyx* from computing all of the eigenvectors between *vl* and *vu*. The number of eigenvectors computed is returned in *nz*.

If $(\text{mod}(\text{info}/8, 2).\text{ne.}0)$, then *p?stebz* failed to compute eigenvalues.

If $(\text{mod}(\text{info}/16, 2).\text{ne.}0)$, then *B* was not positive definite. *ifail*(1) indicates the order of the smallest minor which is not positive definite.

ScaLAPACK Auxiliary, Utility, and Redistribution/Copy Routines

5

This chapter describes the Intel® Math Kernel Library implementation of ScaLAPACK [Auxiliary Routines](#), [Utility Functions and Routines](#), and [Matrix Redistribution/Copy Routines](#). The library includes routines for both real and complex data.

NOTE

ScaLAPACK routines are provided only with Intel® MKL versions for Linux* and Windows* OSs.

Routine naming conventions, mathematical notation, and matrix storage schemes used for ScaLAPACK auxiliary and utility routines are the same as described in previous chapters. Some routines and functions may have combined character codes, such as `sc` or `dz`. For example, the routine `psum1` uses a complex input array and returns a real value.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Auxiliary Routines

ScaLAPACK Auxiliary Routines

Routine Name	Data Types	Description
b?laapp	s, d	Multiplies a matrix with an orthogonal matrix.
b?laexc	s, d	Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.
b?treqc	s, d	Reorders the Schur factorization of a general matrix.
p?lacgv	c, z	Conjugates a complex vector.
p?max1	c, z	Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS <code>pamax</code> , but using the absolute value to the real part).
pmpcol	s, d	<i>Finds the collaborators of a process.</i>
pmpim2	s, d	Computes the eigenpair range assignments for all processes.

Routine Name	Data Types	Description
?combamax1	c, z	Finds the element with maximum real part absolute value and its corresponding global index.
p?sum1	sc, dz	Forms the 1-norm of a complex vector similar to Level 1 PBLAS p?asum, but using the true absolute value.
p?dbtrsv	s, d, c, z	Computes an LU factorization of a general tridiagonal matrix with no pivoting. The routine is called by p?dbtrs.
p?dttrsv	s, d, c, z	Computes an LU factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by p?dttrs.
p?gebal	s, d	Balances a general real matrix.
p?gebd2	s, d, c, z	Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).
p?gehd2	s, d, c, z	Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).
p?gelq2	s, d, c, z	Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).
p?geql2	s, d, c, z	Computes a QL factorization of a general rectangular matrix (unblocked algorithm).
p?geqr2	s, d, c, z	Computes a QR factorization of a general rectangular matrix (unblocked algorithm).
p?gerq2	s, d, c, z	Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).
p?getf2	s, d, c, z	Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).
p?labrd	s, d, c, z	Reduces the first nb rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.
p?lacon	s, d, c, z	Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.
p?laconsb	s, d	Looks for two consecutive small subdiagonal elements.
p?lacp2	s, d, c, z	Copies all or part of a distributed matrix to another distributed matrix.
p?lacp3	s, d	Copies from a global parallel array into a local replicated array or vice versa.
p?lacpy	s, d, c, z	Copies all or part of one two-dimensional array to another.
p?laevswp	s, d, c, z	Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.
p?lahrd	s, d, c, z	Reduces the first nb columns of a general rectangular matrix A so that elements below the k^{th} subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.

Routine Name	Data Types	Description
p?laect	s, d, c, z	Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).
p?lamve	s, d	Copies all or part of one two-dimensional distributed array to another.
p?lange	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.
p?lanhs	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.
p?lansy, p?lanhe	s, d, c, z/c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a real symmetric or complex Hermitian matrix.
p?lantr	s, d, c, z	Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.
p?lapiv	s, d, c, z	Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.
p?laqge	s, d, c, z	Scales a general rectangular matrix, using row and column scaling factors computed by p?geequ.
p?laqr0	s, d	Computes the eigenvalues of a Hessenberg matrix and optionally returns the matrices from the Schur decomposition.
p?laqr1	s, d	Sets a scalar multiple of the first column of the product of a 2-by-2 or 3-by-3 matrix and specified shifts.
p?laqr2	s, d	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
p?laqr3	s, d	Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).
p?laqr4	s, d	Computes the eigenvalues of a Hessenberg matrix, and optionally computes the matrices from the Schur decomposition.
p?laqr5	s, d	Performs a single small-bulge multi-shift QR sweep.
p?laqsy	s, d, c, z	Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ.
p?lared1d	s, d	Redistributes an array assuming that the input array <i>bycol</i> is distributed across rows and that all process columns contain the same copy of <i>bycol</i> .
p?lared2d	s, d	Redistributes an array assuming that the input array <i>byrow</i> is distributed across columns and that all process rows contain the same copy of <i>byrow</i> .
p?larf	s, d, c, z	Applies an elementary reflector to a general rectangular matrix.

Routine Name	Data Types	Description
p?larfb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.
p?larfc	c, z	Applies the conjugate transpose of an elementary reflector to a general matrix.
p?larfg	s, d, c, z	Generates an elementary reflector (Householder matrix).
p?larft	s, d, c, z	Forms the triangular vector T of a block reflector $H=I-VT\bar{V}^H$
p?larz	s, d, c, z	Applies an elementary reflector as returned by p?tzrzf to a general matrix.
p?larzb	s, d, c, z	Applies a block reflector or its transpose/conjugate-transpose as returned by p?tzrzf to a general matrix.
p?larzc	c, z	Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by p?tzrzf to a general matrix.
p?larzt	s, d, c, z	Forms the triangular factor T of a block reflector $H=I-VT\bar{V}^H$ as returned by p?tzrzf.
p?lascl	s, d, c, z	Multiplies a general rectangular matrix by a real scalar defined as $C_{\text{to}}/C_{\text{from}}$.
p?laset	s, d, c, z	Initializes the off-diagonal elements of a matrix to α and the diagonal elements to β .
p?lasmsub	s, d	Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.
p?lassq	s, d, c, z	Updates a sum of squares represented in scaled form.
p?laswp	s, d, c, z	Performs a series of row interchanges on a general rectangular matrix.
p?latra	s, d, c, z	Computes the trace of a general square distributed matrix.
p?latrd	s, d, c, z	Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.
p?latrz	s, d, c, z	Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.
p?lauu2	s, d, c, z	Computes the product UU^H or L^HL , where U and L are upper or lower triangular matrices (local unblocked algorithm).
p?laum	s, d, c, z	Computes the product UU^H or L^HL , where U and L are upper or lower triangular matrices.
p?lawil	s, d	Forms the Wilkinson transform.
p?org2l/p?ung2l	s, d, c, z	Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by p?geqlf (unblocked algorithm).
p?org2r/p?ung2r	s, d, c, z	Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by p?geqrf (unblocked algorithm).
p?orgl2/p?ungl2	s, d, c, z	Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf (unblocked algorithm).

Routine Name	Data Types	Description
p?orgr2/p?ungr2	s, d, c, z	Generates all or part of the orthogonal/unitary matrix Q from an RQ factorization determined by p?gerqf (unblocked algorithm).
p?orm2l/p?unm2l	s, d, c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by p?geqlf (unblocked algorithm).
p?orm2r/p?unm2r	s, d, c, z	Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by p?geqr (unblocked algorithm).
p?orml2/p?unml2	s, d, c, z	Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by p?gelqf (unblocked algorithm).
p?ormr2/p?unmr2	s, d, c, z	Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by p?gerqf (unblocked algorithm).
p?pbtrs	s, d, c, z	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by p?pbtrf.
p?pttrs	s, d, c, z	Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by p?pttrf.
p?potf2	s, d, c, z	Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).
p?rot	s, d	Applies a planar rotation to two distributed vectors.
p?rscl	s, d, cs, zd	Multiplies a vector by the reciprocal of a real scalar.
p?systs2/p?hegs2	s, d, c, z	Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf (local unblocked algorithm).
p?syt2/p?hetd2	s, d, c, z	Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).
p?trord	s, d	Reorders the Schur factorization of a general matrix.
p?trsen	s, d	Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.
p?trti2	s, d, c, z	Computes the inverse of a triangular matrix (local unblocked algorithm).
?lamsh	s, d	Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.
?laqr6	s, d	Performs a single small-bulge multi-shift QR sweep collecting the transformations.
?lar1va	s, d	<i>Computes scaled eigenvector corresponding to given eigenvalue.</i>
?laref	s, d	Applies Householder reflectors to matrices on either their rows or columns.
?larrb2	s, d	Provides limited bisection to locate eigenvalues for more accuracy.
?larrd2	s, d	Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.

Routine Name	Data Types	Description
?larre2	s, d	Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.
?larre2a	s, d	<i>Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.</i>
?larrf2	s, d	Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.
?larrv2	s, d	Computes the eigenvectors of the tridiagonal matrix $T = L^*D*L^T$ given L , D and the eigenvalues of L^*D*L^T .
?lasorte	s, d	Sorts eigenpairs by real and complex data types.
?lasrt2	s, d	Sorts numbers in increasing or decreasing order.
?stegr2	s, d	Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.
?stegr2a	s, d	Computes selected eigenvalues and initial representations needed for eigenvector computations.
?stegr2b	s, d	From eigenvalues and initial representations computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors.
?stein2	s, d	Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.
?dbtf2	s, d, c, z	Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).
?dbtrf	s, d, c, z	Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).
?dttrf	s, d, c, z	Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).
?dttrsv	s, d, c, z	Solves a general tridiagonal system of linear equations using the LU factorization computed by ?dttrf.
?pttrsv	s, d, c, z	Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the LDL^H factorization computed by ?pttrf.
?steqr2	s, d	Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

b?laapp*Multiplies a matrix with an orthogonal matrix.***Syntax****Fortran:**

```
call bslaapp( iside, m, n, nb, a, lda, nitraf, itraf, dtraf, work )
call bdlaapp( iside, m, n, nb, a, lda, nitraf, itraf, dtraf, work )
```

Include Files

- C: mkl_scalapack.h

Description

b?laapp computes

$$B = Q^T A \text{ or } B = A Q$$

where A is an m -by- n matrix and Q is an orthogonal matrix represented by the parameters in the arrays *itraf* and *dtraf* as described in [b?trexc](#).

This is an auxiliary routine called by [p?trord](#).

Input Parameters

<i>iside</i>	INTEGER	Specifies whether Q multiplies A from the left or right as follows: = 0: compute $B = Q^T A$; = 1: compute $B = A Q$.
<i>m</i>	INTEGER	The number of rows of A .
<i>n</i>	INTEGER	The number of columns of A .
<i>nb</i>	INTEGER	If <i>iside</i> = 0, the Q is applied block column-wise to the rows of A and <i>nb</i> specifies the maximal width of the block columns. If <i>iside</i> = 1, this variable is not referenced.
<i>a</i>	REAL for bslaapp DOUBLE PRECISION for bdlaapp	Array, dimension (<i>lda</i> , <i>n</i>) On entry, the matrix A .
<i>lda</i>	INTEGER	The leading dimension of the array <i>a</i> . <i>lda</i> $\geq \max(1,n)$.
<i>nitraf</i>	INTEGER	Length of the array <i>itraf</i> . <i>nitraf</i> ≥ 0 .
<i>itraf</i>	INTEGER array, length <i>nitraf</i>	List of parameters for representing the transformation matrix Q , see b?trexc .
<i>work</i>	(workspace) REAL array, dimension (<i>n</i>)	

OUTPUT Parameters

<i>a</i>	<i>a</i> is overwritten by <i>B</i> .
<i>dtraf</i>	REAL for bslaapp DOUBLE PRECISION for bdlaapp
	Array, length <i>k</i> . If <i>iside</i> =0, <i>k</i> = 3*(<i>n</i> / <i>nb</i>). If <i>iside</i> =1, <i>k</i> = 3* <i>nitrif</i> .
	List of parameters for representing the transformation matrix <i>Q</i> , see b?trexc

b?laexc

Swaps adjacent diagonal blocks of a real upper quasi-triangular matrix in Schur canonical form, by an orthogonal similarity transformation.

Syntax

Fortran:

```
call bslaexc( n, t, ldt, j1, n1, n2, itraf, dtraf, work, info )
call bdlaexc( n, t, ldt, j1, n1, n2, itraf, dtraf, work, info )
```

Include Files

- C: mkl_scalapack.h

Description

`b?laexc` swaps adjacent diagonal blocks T_{11} and T_{22} of order 1 or 2 in an upper quasi-triangular matrix T by an orthogonal similarity transformation.

In contrast to the LAPACK routine `?laexc`, the orthogonal transformation matrix Q is not explicitly constructed but represented by parameters contained in the arrays *itraf* and *dtraf*. See the description of [b?trexc](#) for more details.

T must be in Schur canonical form, that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

<i>n</i>	INTEGER The order of the matrix T . $n \geq 0$.
<i>t</i>	REAL for bslaexc DOUBLE PRECISION for bdlaexc Array, dimension (<i>ldt,n</i>) The upper quasi-triangular matrix T , in Schur canonical form.
<i>ldt</i>	INTEGER

The leading dimension of the array t . $\text{ldt} \geq \max(1, n)$.

j1 INTEGER

The index of the first row of the first block T_{11} .

n1 INTEGER

The order of the first block T_{11} . $n1 = 0, 1$ or 2 .

n2 INTEGER

The order of the second block T_{22} . $n2 = 0, 1$ or 2 .

work REAL for bsblaexc

DOUBLE PRECISION for bdlaexc

(Workspace) array, dimension (n)

OUTPUT Parameters

t The updated matrix T , in Schur canonical form.

itraf INTEGER array, length k , where

$k = 1$, if $n1+n2 = 2$;

$k = 2$, if $n1+n2 = 3$;

$k = 4$, if $n1+n2 = 4$.

List of parameters for representing the transformation matrix Q , see [b?trexc](#).

dtraf REAL for bsblaexc

DOUBLE PRECISION for bdlaexc

Array, length k , where

$k = 2$, if $n1+n2 = 2$;

$k = 5$, if $n1+n2 = 3$;

$k = 10$, if $n1+n2 = 4$.

List of parameters for representing the transformation matrix Q , see [b?trexc](#).

info INTEGER

= 0: successful exit

= 1: the transformed matrix T would be too far from Schur form; the blocks are not swapped and T and Q are unchanged.

b?trexc

Reorders the Schur factorization of a general matrix.

Syntax

Fortran:

```
call bstrexc( n, t, ldt, ifst, ilst, nitraf, itraf, ndtraf, dtraf, work, info )
call bdtrexc( n, t, ldt, ifst, ilst, nitraf, itraf, ndtraf, dtraf, work, info )
```

Include Files

- C: mkl_scalapack.h

Description

`b?trexc` reorders the real Schur factorization of a real matrix $A = Q * T * Q^T$, so that the diagonal block of T with row index `ifst` is moved to row `ilst`.

The real Schur form T is reordered by an orthogonal similarity transformation $Z^T * T * Z$. In contrast to the LAPACK routine `?trexc`, the orthogonal matrix Z is not explicitly constructed but represented by parameters contained in the arrays `itraf` and `dtraf`. See Application Notes for further details.

T must be in Schur canonical form (as returned by `?hseqr`), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks; each 2-by-2 diagonal block has its diagonal elements equal and its off-diagonal elements of opposite sign.

Input Parameters

<code>n</code>	INTEGER	
		The order of the matrix T . $n \geq 0$.
<code>t</code>	REAL for <code>bstrexc</code> DOUBLE PRECISION for <code>bdtrexc</code>	
		Array, dimension (<code>ldt,n</code>)
		The upper quasi-triangular matrix T , in Schur canonical form.
<code>ldt</code>	INTEGER	
		The leading dimension of the array <code>t</code> . $ldt \geq \max(1,n)$.
<code>ifst, ilst</code>	INTEGER	
		Specify the reordering of the diagonal blocks of T . The block with row index <code>ifst</code> is moved to row <code>ilst</code> , by a sequence of transpositions between adjacent blocks.
<code>nitraf</code>	INTEGER	
		Length of the array <code>itraf</code> .
		As a minimum requirement, $nitraf \geq \max(1, ilst-ifst)$.
		If there are 2-by-2 blocks in <code>t</code> then <code>nitraf</code> must be larger; a safe choice is $nitraf \geq \max(1,2* ilst-ifst)$.
<code>ndtraf</code>	INTEGER	
		Length of the array <code>dtraf</code> .
		As a minimum requirement, $ndtraf \geq \max(1,2* ilst-ifst)$.
		If there are 2-by-2 blocks in <code>t</code> then <code>ndtraf</code> must be larger; a safe choice is $ndtraf \geq \max(1,5* ilst-ifst)$.
<code>work</code>	REAL for <code>bstrexc</code>	

DOUBLE PRECISION for bdtrexc
 (Workspace) array, dimension (n)

OUTPUT Parameters

t	On exit, the reordered upper quasi-triangular matrix, in Schur canonical form.
$ifst, ilst$	If $ifst$ pointed on entry to the second row of a 2-by-2 block, it is changed to point to the first row; $ilst$ always points to the first row of the block in its final position (which may differ from its input value by +1 or -1). $1 \leq ifst \leq n; 1 \leq ilst \leq n.$
$nitraf$	Actual length of the array $itraf$.
$itraf$	INTEGER array, length $nitraf$ List of parameters for representing the transformation matrix Z . See Application Notes for further details.
$ndtraf$	Actual length of the array $dtraf$.
$dtraf$	REAL for bstrexc DOUBLE PRECISION for bdtrexc Array, length $ndtraf$ List of parameters for representing the transformation matrix Z . See Application Notes for further details.
$info$	INTEGER = 0: successful exit < 0: if $info = -i$, the i -th argument had an illegal value = 1: two adjacent blocks were too close to swap (the problem is very ill-conditioned); t may have been partially reordered, and $ilst$ points to the first row of the current position of the block being moved. = 2: the 2 by 2 block to be reordered split into two 1 by 1 blocks and the second block failed to swap with an adjacent block. $ilst$ points to the first row of the current position of the whole block being moved.

Application Notes

The orthogonal transformation matrix Z is a product of $nitraf$ elementary orthogonal transformations. The parameters defining these transformations are stored in the arrays $itraf$ and $dtraf$ as follows:

Consider the i -th transformation acting on rows/columns $pos, pos+1, \dots$ If this transformation is

- a Givens rotation with cosine c and sine s then

$$itraf(i) = pos, dtraf(i) = c, dtraf(i+1) = s;$$

- a Householder reflector $H = I - \tau * v * v'$ with $v = [1; v2; v3]$ then

$$itraf(i) = n + pos, dtraf(i) = \tau, dtraf(i+1) = v2, dtraf(i+2) = v3;$$

- a Householder reflector $H = I - \tau * v * v'$ with $v = [v1; v2; 1]$ then

$$itraf(i) = 2*n + pos, dtraf(i) = v1, dtraf(i+1) = v2, dtraf(i+2) = \tau;$$

Note that the parameters in *dtraf* are stored consecutively.

p?lacgv

Conjugates a complex vector.

Syntax

Fortran:

```
call pclacgv(n, x, ix, jx, descx, incx)
call pzlacgv(n, x, ix, jx, descx, incx)
```

C:

```
void pclacgv (MKL_INT *n , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pzlacgv (MKL_INT *n , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
```

Include Files

- C: mkl_scalapack.h

Description

The p?lacgv routine conjugates a complex vector $\text{sub}(x)$ of length n , where $\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $\text{incx} = m_x$, and $X(ix:ix+n-1, jx)$ if $\text{incx} = 1$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of the distributed vector $\text{sub}(x)$.
<i>x</i>	(local). COMPLEX for pclacgv COMPLEX*16 for pzlacgv. Pointer into the local memory to an array of size $(\text{lld}_x, *)$. On entry the vector to be conjugated $x(i) = X(ix + (jx-1)*m_x + (i-1)*\text{incx})$, $1 \leq i \leq n$.
<i>ix</i>	(global) INTEGER. The row index in the global array x indicating the first row of $\text{sub}(x)$.
<i>jx</i>	(global) INTEGER. The column index in the global array x indicating the first column of $\text{sub}(x)$.
<i>descx</i>	(global and local) INTEGER. Array, size (<i>dlen_</i>). The array descriptor for the distributed matrix X .
<i>incx</i>	(global) INTEGER. The global increment for the elements of X . Only two values of <i>incx</i> are supported in this version, namely 1 and m_x . <i>incx</i> must not be zero.

Output Parameters

<i>x</i>	(local).
----------	----------

On exit, the conjugated vector.

p?max1

Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS pamax, but using the absolute value to the real part).

Syntax

Fortran:

```
call pcmax1(n, amax, indx, x, ix, jx, descx, incx)
call pzmax1(n, amax, indx, x, ix, jx, descx, incx)
```

C:

```
void pcmax1 (MKL_INT *n , MKL_Complex8 *amax , MKL_INT *indx , MKL_Complex8 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx );
void pzmax1 (MKL_INT *n , MKL_Complex16 *amax , MKL_INT *indx , MKL_Complex16 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx );
```

Include Files

- C: mkl_scalapack.h

Description

The p?max1 routine computes the global index of the maximum element in absolute value of a distributed vector sub(x). The global index is returned in *indx* and the value is returned in *amax*, where sub(x) denotes $X(ix:ix+n-1, jx)$ if *incx* = 1, $X(ix, jx:jx+n-1)$ if *incx* = *m_x*.

Input Parameters

<i>n</i>	(global) pointer to INTEGER. The number of components of the distributed vector sub(x). $n \geq 0$.
<i>x</i>	(local) COMPLEX for pcmax1. COMPLEX*16 for pzmax1 Array containing the local pieces of a distributed matrix of dimension of at least $((jx-1)*m_x+ix+(n-1)*abs(incx))$. This array contains the entries of the distributed vector sub(x).
<i>ix</i>	(global) INTEGER. The row index in the global array X indicating the first row of sub(x).
<i>jx</i>	(global) INTEGER. The column index in the global array X indicating the first column of sub(x)
<i>descx</i>	(global and local) INTEGER. Array, size (<i>dlen_</i>). The array descriptor for the distributed matrix X.

incx (global) INTEGER. The global increment for the elements of X. Only two values of *incx* are supported in this version, namely 1 and *m_x*. *incx* must not be zero.

Output Parameters

amax (global output) pointer to REAL. The absolute value of the largest entry of the distributed vector sub(x) only in the scope of sub(x).

indx (global output) pointer to INTEGER. The global index of the element of the distributed vector sub(x) whose real part has maximum absolute value.

pmpcol

Finds the collaborators of a process.

Syntax

Fortran:

```
call pmpcol( myproc, nprocs, iil, needil, neediu, pmyils, pmyius, colbrt, frstcl,
lastcl )
```

Include Files

- C: mkl_scalapack.h

Description

Using the output from [pmpim2](#) and given the information on eigenvalue clusters, pmpcol finds the collaborators of *myproc*.

Input Parameters

myproc INTEGER
The processor number, $0 \leq \text{myproc} < \text{nprocs}$.

nprocs INTEGER
The total number of processors available.

iil INTEGER
The index of the leftmost eigenvalue in the eigenvalue cluster.

needil INTEGER
The leftmost position in the eigenvalue cluster needed by *myproc*.

neediu INTEGER
The rightmost position in the eigenvalue cluster needed by *myproc*.

pmyils INTEGER array
For each processor p , *pmyils*(p) is the index of the first eigenvalue in the eigenvalue cluster to be computed.
pmyils(p) equals zero if p stays idle.

pmyius INTEGER array
 For each processor p , $pmyius(p)$ is the index of the last eigenvalue in the eigenvalue cluster to be computed.
 $pmyius(p)$ equals zero if p stays idle.

OUTPUT Parameters

colbrt LOGICAL
 TRUE if $myproc$ collaborates.

frstcl, lastcl INTEGER
 First and last collaborator of $myproc$.
 $myproc$ collaborates with:
 $frstcl, \dots, myproc-1, myproc+1, \dots, lastcl$
 If $myproc = frstcl$, there are no collaborators on the left. If $myproc = lastcl$, there are no collaborators on the right.
 If $frstcl = 0$ and $lastcl = nprocs-1$, then $myproc$ collaborates with everybody

pmpim2

Computes the eigenpair range assignments for all processes.

Syntax

Fortran:

```
call pmpim2( il, iu, nprocs, pmyils, pmyius )
```

Include Files

- C: mkl_scalapack.h

Description

pmpim2 is the scheduling subroutine. It computes for all processors the eigenpair range assignments.

Input Parameters

il, iu INTEGER
 The range of eigenpairs to be computed.

nprocs INTEGER
 The total number of processors available.

Output Parameters

pmyils INTEGER array
 For each processor p , $pmyils(p)$ is the index of the first eigenvalue in a cluster to be computed.

$pmyils(p)$ equals zero if p stays idle.

$pmyius$

INTEGER array

For each processor p , $pmyius(p)$ is the index of the last eigenvalue in a cluster to be computed.

$pmyius(p)$ equals zero if p stays idle.

?combamax1

Finds the element with maximum real part absolute value and its corresponding global index.

Syntax

Fortran:

```
call ccombamax1(v1, v2)
call zcombamax1(v1, v2)
```

C:

```
void ccombamax1 (MKL_Complex8 *v1 , MKL_Complex8 *v2 );
void zcombamax1 (MKL_Complex16 *v1 , MKL_Complex16 *v2 );
```

Include Files

- C: mkl_scalapack.h

Description

The ?combamax1 routine finds the element having maximum real part absolute value as well as its corresponding global index.

Input Parameters

v1

(local)

COMPLEX for ccombamax1

COMPLEX*16 for zcombamax1

Array, size 2. The first maximum absolute value element and its global index. $v1(1)=amax$, $v1(2)=indx$.

v2

(local)

COMPLEX for ccombamax1

COMPLEX*16 for zcombamax1

Array, size 2. The second maximum absolute value element and its global index. $v2(1)=amax$, $v2(2)=indx$.

Output Parameters

v1

(local).

The first maximum absolute value element and its global index.

$v1(1)=amax$, $v1(2)=indx$.

p?sum1

Forms the 1-norm of a complex vector similar to Level 1 PBLAS p?asum, but using the true absolute value.

Syntax

Fortran:

```
call pscsum1(n, asum, x, ix, jx, descx, incx)
call pdzsum1(n, asum, x, ix, jx, descx, incx)
```

C:

```
void pscsum1 (MKL_INT *n , float *asum , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
void pdzsum1 (MKL_INT *n , double *asum , MKL_Complex16 *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , MKL_INT *incx );
```

Include Files

- C: mkl_scalapack.h

Description

The p?sum1 routine returns the sum of absolute values of a complex distributed vector sub(*x*) in *asum*, where sub(*x*) denotes $X(ix:ix+n-1, jx:jx)$, if *incx* = 1, $X(ix:ix, jx:jx+n-1)$, if *incx* = *m_x*.

Based on p?asum from the Level 1 PBLAS. The change is to use the 'genuine' absolute value.

Input Parameters

<i>n</i>	(global) pointer to INTEGER. The number of components of the distributed vector sub(<i>x</i>). $n \geq 0$.
<i>x</i>	(local) COMPLEX for pscsum1 COMPLEX*16 for pdzsum1. Array containing the local pieces of a distributed matrix of dimension of at least $((jx-1)*m_x+ix+(n-1)*abs(incx))$. This array contains the entries of the distributed vector sub (<i>x</i>).
<i>ix</i>	(global) INTEGER. The row index in the global array <i>X</i> indicating the first row of sub(<i>x</i>).
<i>jx</i>	(global) INTEGER. The column index in the global array <i>X</i> indicating the first column of sub(<i>x</i>)
<i>descx</i>	(local) INTEGER. Array, size 9. The array descriptor for the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. The global increment for the elements of <i>X</i> . Only two values of <i>incx</i> are supported in this version, namely 1 and <i>m_x</i> .

Output Parameters

<i>asum</i>	(local)
-------------	---------

Pointer to `REAL`. The sum of absolute values of the distributed vector $\text{sub}(x)$ only in its scope.

p?dbtrsv

Computes an LU factorization of a general triangular matrix with no pivoting. The routine is called by p?dbtrs.

Syntax

Fortran:

```
call psdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pddbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pcdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pzdbtrsv(uplo, trans, n, bwl, bwu, nrhs, a, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

C:

```
void psdbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu ,
MKL_INT *nrhs , float *a , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib ,
MKL_INT *descb , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT
*info );

void pddbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu ,
MKL_INT *nrhs , double *a , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *descb , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT
*info );

void pcdbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu ,
MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzdbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu ,
MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16
*work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: `mkl_scalapack.h`

Description

The `p?dbtrsv` routine solves a banded triangular system of linear equations

$$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs) \text{ or}$$

$$A(1:n, ja:ja+n-1)^T * X = B(ib:ib+n-1, 1:nrhs) \text{ (for real flavors); } A(1:n, ja:ja+n-1)^H * X = B(ib:ib+n-1, 1:nrhs) \text{ (for complex flavors),}$$

where $A(1 :n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Gaussian elimination code `PD@ (dom_pre) BTRF` and is stored in $A(1 :n, ja:ja+n-1)$ and `af`. The matrix stored in $A(1 :n, ja:ja+n-1)$ is either upper or lower triangular according to `uplo`, and the choice of solving $A(1 :n, ja:ja+n-1)$ or $A(1 :n, ja:ja+n-1)^T$ is dictated by the user by the parameter `trans`.

Routine `p?dbtrf` must be called first.

Input Parameters

<code>uplo</code>	(global) CHARACTER. If <code>uplo='U'</code> , the upper triangle of $A(1:n, ja:ja+n-1)$ is stored, if <code>uplo = 'L'</code> , the lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<code>trans</code>	(global) CHARACTER. If <code>trans = 'N'</code> , solve with $A(1:n, ja:ja+n-1)$, if <code>trans = 'C'</code> , solve with conjugate transpose $A(1:n, ja:ja+n-1)$.
<code>n</code>	(global) INTEGER. The order of the distributed submatrix A ; ($n \geq 0$).
<code>bwl</code>	(global) INTEGER. Number of subdiagonals. $0 \leq bwl \leq n-1$.
<code>bwu</code>	(global) INTEGER. Number of subdiagonals. $0 \leq bwu \leq n-1$.
<code>nrhs</code>	(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix B ($nrhs \geq 0$).
<code>a</code>	(local). REAL for <code>psdbtrsv</code> DOUBLE PRECISION for <code>pddbtrsv</code> COMPLEX for <code>pcdbtrsv</code> COMPLEX*16 for <code>pzdbtrsv</code> . Pointer into the local memory to an array with first size $l/d_a \geq (bwl + bwu + 1)$ (stored in <code>desca</code>). On entry, this array contains the local pieces of the n -by- n unsymmetric banded distributed Cholesky factor L or $L^T * A(1 :n, ja:ja+n-1)$. This local portion is stored in the packed banded format used in LAPACK. See the <i>Application Notes</i> below and the ScaLAPACK manual for more detail on the format of distributed matrices.
<code>ja</code>	(global) INTEGER. The index in the global array <code>a</code> that points to the start of the matrix to be operated on (which may be either all of A or a submatrix of A).
<code>desca</code>	(global and local) INTEGER array of size (<code>dlen_</code>). if 1d type (<code>dtype_a = 501</code> or <code>502</code>), <code>dlen \geq 7</code> ; if 2d type (<code>dtype_a = 1</code>), <code>dlen \geq 9</code> . The array descriptor for the distributed matrix A . Contains information of mapping of A to memory.
<code>b</code>	(local) REAL for <code>psdbtrsv</code> DOUBLE PRECISION for <code>pddbtrsv</code>

COMPLEX for pcdbtrsv

COMPLEX*16 for pzdbtrsv.

Pointer into the local memory to an array of local lead dimension $lld_b \geq nb$. On entry, this array contains the local pieces of the right-hand sides $B(ib:ib+n-1, 1:nrhs)$.

ib (global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

descb (global and local) INTEGER array of size (*dlen_*).
 if 1d type (*dtype_b* = 502), *dlen* ≥ 7 ;
 if 2d type (*dtype_b* = 1), *dlen* ≥ 9 . The array descriptor for the distributed matrix *B*. Contains information of mapping *B* to memory.

laf (local)
 INTEGER. Size of user-input Auxiliary Filling space *af*.
laf must be $\geq nb * (bwl + bwu) + 6 * \max(bwl, bwu) * \max(bwl, bwu)$. If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*(1).

work (local).
 REAL for psdbtrsv
 DOUBLE PRECISION for pddbtrsv
 COMPLEX for pcdbtrsv
 COMPLEX*16 for pzdbtrsv.
 Temporary workspace. This space may be overwritten in between calls to routines.
work must be the size given in *lwork*.

lwork (local or global) INTEGER.
 Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.
 $lwork \geq \max(bwl, bwu) * nrhs$.

Output Parameters

a (local).
 This local portion is stored in the packed banded format used in LAPACK. Please see the ScaLAPACK manual for more detail on the format of distributed matrices.

b On exit, this contains the local piece of the solutions distributed matrix *X*.

af (local).
 REAL for psdbtrsv
 DOUBLE PRECISION for pddbtrsv

COMPLEX for pcdbtrsv
 COMPLEX*16 for pzdbtrsv.

Auxiliary Filling Space. Filling is created during the factorization routine p?dbtrf and this is stored in af. If a linear system is to be solved using p?dbtrf after the factorization routine, af must not be altered after the factorization.

work

On exit, work(1) contains the minimal lwork.

info

(local).

INTEGER. If info = 0, the execution is successful.

< 0: If the i-th argument is an array and the j-entry had an illegal value, then info = - (i*100+j), if the i-th argument is a scalar and had an illegal value, then info = -i.

p?dttrs

Computes an LU factorization of a general band matrix, using partial pivoting with row interchanges.
 The routine is called by p?dttrs.

Syntax

Fortran:

```
call psdttrs(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pddttrs(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pcdttrs(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)

call pzdttrs(uplo, trans, n, nrhs, dl, d, du, ja, desca, b, ib, descb, af, laf,
work, lwork, info)
```

C:

```
void psdttrs (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , float *dl ,
float *d , float *du , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT
*descb , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pddttrs (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , double *dl ,
double *d , double *du , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *descb , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT
*info );

void pcdttrs (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8
*dl , MKL_Complex8 *d , MKL_Complex8 *du , MKL_INT *ja , MKL_INT *desca , MKL_Complex8
*b , MKL_INT *ib , MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8
*work , MKL_INT *lwork , MKL_INT *info );

void pzdttrs (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16
*dl , MKL_Complex16 *d , MKL_Complex16 *du , MKL_INT *ja , MKL_INT *desca ,
MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?dttrs` routine solves a tridiagonal triangular system of linear equations

$A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs)$ or

$A(1:n, ja:ja+n-1)^T * X = B(ib:ib+n-1, 1:nrhs)$ for real flavors; $A(1:n, ja:ja+n-1)^H * X = B(ib:ib+n-1, 1:nrhs)$ for complex flavors,

where $A(1:n, ja:ja+n-1)$ is a tridiagonal matrix factor produced by the Gaussian elimination code `PS@(dom_pre)TTRF` and is stored in $A(1:n, ja:ja+n-1)$ and af .

The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to `uplo`, and the choice of solving $A(1:n, ja:ja+n-1)$ or $A(1:n, ja:ja+n-1)^T$ is dictated by the user by the parameter `trans`.

Routine `p?dttrf` must be called first.

Input Parameters

`uplo`

(global) CHARACTER.

If `uplo='U'`, the upper triangle of $A(1:n, ja:ja+n-1)$ is stored,
if `uplo = 'L'`, the lower triangle of $A(1:n, ja:ja+n-1)$ is stored.

`trans`

(global) CHARACTER.

If `trans = 'N'`, solve with $A(1:n, ja:ja+n-1)$,
if `trans = 'C'`, solve with conjugate transpose $A(1:n, ja:ja+n-1)$.

`n`

(global) INTEGER. The order of the distributed submatrix A ; ($n \geq 0$).

`nrhs`

(global) INTEGER. The number of right-hand sides; the number of columns of the distributed submatrix $B(ib:ib+n-1, 1:nrhs)$. ($nrhs \geq 0$).

`d1`

(local).

REAL for `psdttrs`
DOUBLE PRECISION for `pddttrs`
COMPLEX for `pcdttrs`
COMPLEX*16 for `pzdttrs`.

Pointer to local part of global vector storing the lower diagonal of the matrix.

Globally, `d1(1)` is not referenced, and `d1` must be aligned with `d`.

Must be of size $\geq \text{desca}(\text{nb}__)$.

`d`

(local).

REAL for `psdttrs`
DOUBLE PRECISION for `pddttrs`
COMPLEX for `pcdttrs`

COMPLEX*16 **for** pzdttrs.

Pointer to local part of global vector storing the main diagonal of the matrix.

du
(local).

REAL **for** psdttrs
DOUBLE PRECISION **for** pddttrs
COMPLEX **for** pcdttrs
COMPLEX*16 **for** pzdttrs.

Pointer to local part of global vector storing the upper diagonal of the matrix.

Globally, *du*(*n*) is not referenced, and *du* must be aligned with *d*.

ja
(global) INTEGER. The index in the global array *a* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca
(global and local) INTEGER array of size (*dlen*_).
if 1d type (*dtype_a* = 501 or 502), *dlen* \geq 7;
if 2d type (*dtype_a* = 1), *dlen* \geq 9.
The array descriptor for the distributed matrix *A*. Contains information of mapping of *A* to memory.

b
(local)
REAL **for** psdttrs
DOUBLE PRECISION **for** pddttrs
COMPLEX **for** pcdttrs
COMPLEX*16 **for** pzdttrs.
Pointer into the local memory to an array of local lead dimension *lld_b* \geq *nb*. On entry, this array contains the local pieces of the right-hand sides *B*(*ib*:*ib+n-1*, 1 :*nrhs*).

ib
(global) INTEGER. The row index in the global array *b* that points to the first row of the matrix to be operated on (which may be either all of *b* or a submatrix of *B*).

descb
(global and local) INTEGER array of size (*dlen*_).
if 1d type (*dtype_b* = 502), *dlen* \geq 7;
if 2d type (*dtype_b* = 1), *dlen* \geq 9.
The array descriptor for the distributed matrix *B*. Contains information of mapping *B* to memory.

laf
(local).
INTEGER.
Size of user-input Auxiliary Filling space *af*.

laf must be $\geq 2 * (nb+2)$. If *laf* is not large enough, an error code is returned and the minimum acceptable size will be returned in *af*(1).

work (local).

REAL for psdttrs v

DOUBLE PRECISION for pddttrs v

COMPLEX for pcdttrs v

COMPLEX*16 for pzdttrs v.

Temporary workspace. This space may be overwritten in between calls to routines.

work must be the size given in *lwork*.

lwork

(local or global) INTEGER.

Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

lwork $\geq 10 * npcol + 4 * nrhs$.

Output Parameters

d1 (local).

On exit, this array contains information containing the factors of the matrix.

d

On exit, this array contains information containing the factors of the matrix. Must be of size $\geq desca(nb_)$.

b

On exit, this contains the local piece of the solutions distributed matrix X.

af (local).

REAL for psdttrs v

DOUBLE PRECISION for pddttrs v

COMPLEX for pcdttrs v

COMPLEX*16 for pzdttrs v.

Auxiliary Filling Space. Filling is created during the factorization routine p?dttrf and this is stored in *af*. If a linear system is to be solved using p?dttrs after the factorization routine, *af* must not be altered after the factorization.

work

On exit, *work*(1) contains the minimal *lwork*.

info

(local). INTEGER.

If *info*=0, the execution is successful.

if *info*< 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?gebal

Balances a general real matrix.

Syntax

Fortran:

```
call psgebal( job, n, a, desca, ilo, ihi, scale, info )
call pdgebal( job, n, a, desca, ilo, ihi, scale, info )
```

Include Files

- C: mkl_scalapack.h

Description

p?gebal balances a general real matrix A . This involves, first, permuting A by a similarity transformation to isolate eigenvalues in the first ilo to $ihi-1$ and last $ihi+1$ to n elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns ilo to ihi to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrix, and improve the accuracy of the computed eigenvalues and/or eigenvectors.

Input Parameters

<i>job</i>	(global) CHARACTER*1
	Specifies the operations to be performed on <i>a</i> :
	= 'N': none: simply set $ilo = 1$, $ihi = n$, $scale(i) = 1.0$ for $i = 1, \dots, n$;
	= 'P': permute only;
	= 'S': scale only;
	= 'B': both permute and scale.
<i>n</i>	(global) INTEGER
	The order of the matrix A ($n \geq 0$).
<i>a</i>	REAL for psgebal DOUBLE PRECISION for pdgebal (local) array, dimension ($desca(1\text{ld}_\text{c}), LOC_c(n)$) The input matrix A .
<i>desca</i>	(global and local) INTEGER array of dimension <i>dlen_</i> . The array descriptor for the distributed matrix A .

OUTPUT Parameters

<i>a</i>	On exit, <i>a</i> is overwritten by the balanced matrix. If <i>job</i> = 'N', <i>a</i> is not referenced. See Notes for further details.
<i>ilo, ihi</i>	(global) INTEGER

ilo and *ih*i are set to integers such that on exit $a(i,j) = 0$ if $i > j$ and $j = 1, \dots, ilo-1$ or $i = ih+1, \dots, n$.

If *job* = 'N' or 'S', *ilo* = 1 and *ih*i = *n*.

scale

REAL for psgebal

DOUBLE PRECISION for pdgebal

(global) array, dimension (*n*)

Details of the permutations and scaling factors applied to *a*. If *p(j)* is the index of the row and column interchanged with row and column *j* and *d(j)* is the scaling factor applied to row and column *j*, then

scale(j) = *p(j)* for *j* = 1, ..., *ilo*-1

= *d(j)* for *j* = *ilo*, ..., *ih*i

The order in which the interchanges are made is *n* to *ih*i+1, then 1 to *ilo*-1.

info

(global) INTEGER

= 0: successful exit.

< 0: if INFO = -*i*, the *i*-th argument had an illegal value.

Application Notes

The permutations consist of row and column interchanges which put the matrix in the form

$$PAP = \begin{pmatrix} T_1 & X & Y \\ 0 & B & Z \\ 0 & 0 & T_2 \end{pmatrix}$$

where *T*1 and *T*2 are upper triangular matrices whose eigenvalues lie along the diagonal. The column indices *ilo* and *ih*i mark the starting and ending columns of the submatrix *B*. Balancing consists of applying a diagonal similarity transformation *D*⁻¹ *B* *D* to make the 1-norms of each row of *B* and its corresponding column nearly equal. The output matrix is

$$\begin{pmatrix} T_1 & XD & Y \\ 0 & D^{-1}BD & D^{-1}Z \\ 0 & 0 & T_2 \end{pmatrix}$$

Information about the permutations *P* and the diagonal matrix *D* is returned in the vector *scale*.

p?gebd2

Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).

Syntax

Fortran:

```
call psgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pdgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pcgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
call pzgebd2(m, n, a, ia, ja, desca, d, e, tauq, taup, work, lwork, info)
```

C:

```
void psgebd2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *d , float *e , float *tauq , float *taup , float *work , MKL_INT
*lwork , MKL_INT *info );
void pdgebd2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *d , double *e , double *tauq , double *taup , double *work ,
MKL_INT *lwork , MKL_INT *info );
void pcgebd2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *d , float *e , MKL_Complex8 *tauq , MKL_Complex8 *taup ,
MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );
void pzgebd2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *d , double *e , MKL_Complex16 *tauq , MKL_Complex16 *taup ,
MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?gebd2 routine reduces a real/complex general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form B by an orthogonal/unitary transformation:

$Q^T \text{sub}(A) P = B$.

If $m \geq n$, B is the upper bidiagonal; if $m < n$, B is the lower bidiagonal.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). REAL for psgebd2 DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

Pointer into the local memory to an array of size ($l1d_a$, $LOCc(ja+n-1)$).

On entry, this array contains the local pieces of the general distributed matrix sub(A).

ia , ja

(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

$desca$

(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix A .

$work$

(local).

REAL for psgebd2

DOUBLE PRECISION for pdgebd2

COMPLEX for pcgebd2

COMPLEX*16 for pzgebd2.

This is a workspace array of size ($lwork$).

$lwork$

(local or global) INTEGER.

The size of the array $work$.

$lwork$ is local input and must be at least $lwork \geq \max(mpa0, nqa0)$,

where $nb = mb_a = nb_a$, $iroffa = \text{mod}(ia-1, nb)$,

$iarow = \text{indxg2p}(ia, nb, myrow, rsrc_a, nprow)$,

$iacol = \text{indxg2p}(ja, nb, mycol, csrc_a, npcol)$,

$mpa0 = \text{numroc}(m+iroffa, nb, myrow, iarow, nprow)$,

$nqa0 = \text{numroc}(n+icoffa, nb, mycol, iacol, npcol)$.

`indxg2p` and `numroc` are ScaLAPACK tool functions; $myrow$, $mycol$, $nprow$, and $npcol$ can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pixerbla`.

Output Parameters

a

(local).

On exit, if $m \geq n$, the diagonal and the first superdiagonal of sub(A) are overwritten with the upper bidiagonal matrix B ; the elements below the diagonal, with the array τ_{aq} , represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the first superdiagonal, with the array τ_{ap} , represent the orthogonal matrix P as a product of elementary reflectors. If $m < n$, the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix B ; the elements below the first subdiagonal, with the array τ_{aq} , represent the

orthogonal/unitary matrix Q as a product of elementary reflectors, and the elements above the diagonal, with the array τ_{up} , represent the orthogonal matrix P as a product of elementary reflectors. See *Applications Notes* below.

d

(local)

```
REAL for psgebd2
DOUBLE PRECISION for pdgebd2
COMPLEX for pcgebd2
COMPLEX*16 for pzgebd2.
```

Array, size $LOCc(ja+\min(m,n)-1)$ **if** $m \geq n$; $LOCr(ia+\min(m,n)-1)$ otherwise. The distributed diagonal elements of the bidiagonal matrix B : $d(i) = a(i,i)$. d is tied to the distributed matrix A .

e

(local)

```
REAL for psgebd2
DOUBLE PRECISION for pdgebd2
COMPLEX for pcgebd2
COMPLEX*16 for pzgebd2.
```

Array, size $LOCc(ja+\min(m,n)-1)$ **if** $m \geq n$; $LOCr(ia+\min(m,n)-2)$ otherwise. The distributed diagonal elements of the bidiagonal matrix B :
if $m \geq n$, $e(i) = a(i, i+1)$ **for** $i = 1, 2, \dots, n-1$;
if $m < n$, $e(i) = a(i+1, i)$ **for** $i = 1, 2, \dots, m-1$. e is tied to the distributed matrix A .

tauq

(local).

```
REAL for psgebd2
DOUBLE PRECISION for pdgebd2
COMPLEX for pcgebd2
COMPLEX*16 for pzgebd2.
```

Array, size $LOCc(ja+\min(m,n)-1)$. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix Q . $tauq$ is tied to the distributed matrix A .

taup

(local).

```
REAL for psgebd2
DOUBLE PRECISION for pdgebd2
COMPLEX for pcgebd2
COMPLEX*16 for pzgebd2.
```

Array, size $LOCr(ia+\min(m,n)-1)$. The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix P . $taup$ is tied to the distributed matrix A .

work

On exit, $work(1)$ returns the minimal and optimal $lwork$.

info (local)
 INTEGER.
 If *info* = 0, the execution is successful.
 if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = - (*i**100+*j*), if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

Application Notes

The matrices *Q* and *P* are represented as products of elementary reflectors:

If $m \geq n$,

$$Q = H(1) * H(2) * \dots * H(n), \text{ and } P = G(1) * G(2) * \dots * G(n-1)$$

Each *H(i)* and *G(i)* has the form:

$$H(i) = I - tauq * v * v', \text{ and } G(i) = I - taup * u * u',$$

where *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors. $v(1: i-1) = 0$, $v(i) = 1$, and $v(i+1:m)$ is stored on exit in

$$A(ia+i-ia+m-1, a+i-1);$$

$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+2:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;

tauq is stored in *TAUQ(ja+i-1)* and *taup* in *TAUP(ia+i-1)*.

If $m < n$,

$v(1: i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in $A(ia+i+1: ia+m-1, ja+i-1)$;

$u(1: i-1) = 0$, $u(i) = 1$, and $u(i+1 :n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$;

tauq is stored in *TAUQ(ja+i-1)* and *taup* in *TAUP(ia+i-1)*.

The contents of sub(*A*) on exit are illustrated by the following examples:

$$\begin{array}{l} m = 6 \text{ and } n = 5(m > n) : \\ \left[\begin{array}{cccccc} d & e & u1 & u1 & u1 \\ v1 & d & e & u2 & u2 \\ v1 & v2 & d & e & u3 \\ v1 & v2 & v3 & d & e \\ v1 & v2 & v3 & v4 & d \\ v1 & v2 & v3 & v4 & v5 \end{array} \right] \end{array} \quad \begin{array}{l} m = 5 \text{ and } n = 6(m < n) : \\ \left[\begin{array}{cccccc} d & u1 & u1 & u1 & u1 & u1 \\ e & d & u2 & u2 & u2 & u2 \\ v1 & e & d & u3 & u3 & u3 \\ v1 & v2 & e & d & u4 & u4 \\ v1 & v2 & v3 & e & d & u5 \end{array} \right] \end{array}$$

where *d* and *e* denote diagonal and off-diagonal elements of *B*, *vi* denotes an element of the vector defining *H(i)*, and *ui* an element of the vector defining *G(i)*.

p?gehd2

Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).

Syntax

Fortran:

```
call psgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pdgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
```

```
call pcgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
call pzgehd2(n, ilo, ihi, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psgehd2 (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT
*info );
void pdgehd2 (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT
*info );
void pcgehd2 (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
void pzgehd2 (MKL_INT *n , MKL_INT *ilo , MKL_INT *ihi , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?gehd2 routine reduces a real/complex general distributed matrix sub(A) to upper Hessenberg form H by an orthogonal/unitary similarity transformation: $Q' \cdot \text{sub}(A) \cdot Q = H$, where $\text{sub}(A) = A(ia+n-1 : ia+ n-1, ja+n-1 : ja+n-1)$.

Input Parameters

n (global) INTEGER. The order of the distributed submatrix A . ($n \geq 0$).

ilo, ihi (global) INTEGER. It is assumed that sub(A) is already upper triangular in rows $ia:ia+ilo-2$ and $ia+ihi:ia+n-1$ and columns $ja:ja+jlo-2$ and $ja+jhi:ja+n-1$. See *Application Notes* for further information.

If $n \geq 0$, $1 \leq ilo \leq ihi \leq n$; otherwise set $ilo = 1$, $ihi = n$.

a (local).

REAL for psgehd2

DOUBLE PRECISION for pdgehd2

COMPLEX for pcgehd2

COMPLEX*16 for pzgehd2.

Pointer into the local memory to an array of size $(\text{lld}_a, \text{LOC}_c(ja+n-1))$.

On entry, this array contains the local pieces of the n -by- n general distributed matrix sub(A) to be reduced.

ia, ja (global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix A , respectively.

<i>desca</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local). REAL for psgehd2 DOUBLE PRECISION for pdgehd2 COMPLEX for pcgehd2 COMPLEX*16 for pzgehd2. This is a workspace array of size (<i>lwork</i>). <i>lwork</i> (local or global) INTEGER. The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq nb + \max(npa0, nb)$, where $nb = mb_a = nb_a$, $iroffa = \text{mod}(ia-1, nb)$, $iarow = \text{indxg2p}(ia, nb, myrow, rsrc_a, nprow)$, $npa0 = \text{numroc}(ihi+iroffa, nb, myrow, iarow, nprow)$. <i>indxg2p</i> and <i>numroc</i> are ScaLAPACK tool functions; <i>myrow</i> , <i>mycol</i> , <i>nprow</i> , and <i>npcol</i> can be determined by calling the subroutine <i>blacs_gridinfo</i> . If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by <i>pxerbla</i> .
Output Parameters	
<i>a</i>	(local). On exit, the upper triangle and the first subdiagonal of sub(<i>A</i>) are overwritten with the upper Hessenberg matrix <i>H</i> , and the elements below the first subdiagonal, with the array <i>tau</i> , represent the orthogonal/unitary matrix <i>Q</i> as a product of elementary reflectors. (see <i>Application Notes</i> below).
<i>tau</i>	(local). REAL for psgehd2 DOUBLE PRECISION for pdgehd2 COMPLEX for pcgehd2 COMPLEX*16 for pzgehd2. Array, size $LOC_c(ja+n-2)$ The scalar factors of the elementary reflectors (see <i>Application Notes</i> below). Elements $ja:ja+ilo-2$ and $ja+ihi:ja+n-2$ of <i>tau</i> are set to zero. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. If <i>info</i> = 0, the execution is successful. if <i>info</i> < 0: If the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = $-(i*100+j)$, if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = $-i$.

Application Notes

The matrix Q is represented as a product of $(ihi-ilo)$ elementary reflectors

$$Q = H(ilo) * H(ilo+1) * \dots * H(ihi-1).$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T,$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1: i)=0$, $v(i+1)=1$ and $v(ihi+1:n)=0$; $v(i+2:ihi)$ is stored on exit in $A(ia+ilo+i:ia+ihi-1, ia+ilo+i-2)$, and τ in $\tau(ia+ilo+i-2)$.

The contents of $A(ia:ia+n-1, ja:ja+n-1)$ are illustrated by the following example, with $n = 7$, $ilo = 2$ and $ihi = 6$:

on entry	on exit
$\begin{bmatrix} a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \end{bmatrix}$	$\begin{bmatrix} a & a & h & h & h & h & a \\ a & h & h & h & h & a & a \\ h & h & h & h & h & h & h \\ v2 & h & h & h & h & h & h \\ v2 & v3 & h & h & h & h & h \\ v2 & v3 & v4 & h & h & h & h \\ a & a & a & a & a & a & a \end{bmatrix}$

where a denotes an element of the original matrix $\text{sub}(A)$, h denotes a modified element of the upper Hessenberg matrix H , and vi denotes an element of the vector defining $H(ja+ilo+i-2)$.

p?gelq2

Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

Fortran:

```
call psgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgelq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psgelq2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdgelq2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcgelq2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );
void pzgelq2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?gelq2` routine computes an LQ factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = L^*Q$.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). REAL for <code>psgelq2</code> DOUBLE PRECISION for <code>pdgelq2</code> COMPLEX for <code>pcgelq2</code> COMPLEX*16 for <code>pzgelq2</code> . Pointer into the local memory to an array of size (<code>lld_a, LOCc(ja+n-1)</code>). On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ which is to be factored.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix <i>A</i> , respectively.
<i>desca</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>work</i>	(local). REAL for <code>psgelq2</code> DOUBLE PRECISION for <code>pdgelq2</code> COMPLEX for <code>pcgelq2</code> COMPLEX*16 for <code>pzgelq2</code> . This is a workspace array of size (<i>lwork</i>).
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $\text{lwork} \geq nq0 + \max(1, mp0)$, where $\text{iroff} = \text{mod}(ia-1, mb_a)$, $\text{icoff} = \text{mod}(ja-1, nb_a)$, $\text{iarow} = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$, $\text{iacol} = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol)$,

```

mp0 = numroc(m+iroff, mb_a, myrow, iarow, nprow),
nq0 = numroc(n+icoff, nb_a, mycol, iacol, npcol),
indxg2p and numroc are ScaLAPACK tool functions; myrow, mycol, nprow,
and npcol can be determined by calling the subroutine blacs_gridinfo.

```

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a	(local).
	On exit, the elements on and below the diagonal of sub(A) contain the m by $\min(m,n)$ lower trapezoidal matrix L (L is lower triangular if $m \leq n$); the elements above the diagonal, with the array tau , represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
tau	(local).
	REAL for psgelq2 DOUBLE PRECISION for pdgelq2 COMPLEX for pcgelq2 COMPLEX*16 for pzgelq2. Array, size $LOCr(ia+\min(m, n)-1)$. This array contains the scalar factors of the elementary reflectors. tau is tied to the distributed matrix A .
$work$	On exit, $work(1)$ returns the minimal and optimal $lwork$.
$info$	(local) INTEGER. If $info = 0$, the execution is successful. if $info < 0$: If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(ia+k-1)*H(ia+k-2)*\dots*H(ia)$ for real flavors, $Q = (H(ia+k-1))^H * (H(ia+k-2))^H * \dots * (H(ia))^H$ for complex flavors,

where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - tau*v*v'$

where tau is a real/complex scalar, and v is a real/complex vector with $v(1: i-1) = 0$ and $v(i) = 1$; $v(i+1: n)$ (for real flavors) or $\text{conjg}(v(i+1: n))$ (for complex flavors) is stored on exit in $A(ia+i-1, ja+i: ja+n-1)$, and tau in $TAU(ia+i-1)$.

p?geql2

Computes a *QL* factorization of a general rectangular matrix (unblocked algorithm).

Syntax

Fortran:

```
call psgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeql2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psgeql2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdgeql2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcgeql2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );
void pzgeql2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?geql2 routine computes a *QL* factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q * L$.

Input Parameters

m (global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).

n (global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).

a (local).

REAL for psgeql2

DOUBLE PRECISION for pdgeql2

COMPLEX for pcgeql2

COMPLEX*16 for pzgeql2.

Pointer into the local memory to an array of size (*lld_a,LOCc(ja+n-1)*).

On entry, this array contains the local pieces of the m -by- n distributed matrix sub(A) which is to be factored.

ia, ja

(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

desca

(global and local) INTEGER array, size (*dlen_*). The array descriptor for the distributed matrix A .

work

(local).

REAL for *psgeql2*

DOUBLE PRECISION for *pdgeql2*

COMPLEX for *pcgeql2*

COMPLEX*16 for *pzgeql2*.

This is a workspace array of size (*lwork*).

lwork

(local or global) INTEGER.

The size of the array *work*.

lwork is local input and must be at least $lwork \geq mp0 + \max(1, nq0)$,

where $iroff = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$,

$iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol)$,

$mp0 = \text{numroc}(m+iroff, mb_a, myrow, iarow, nprow)$,

$nq0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npcol)$,

indxg2p and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nprow*, and *npcol* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

a

(local).

On exit,

if $m \geq n$, the lower triangle of the distributed submatrix $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the n -by- n lower triangular matrix L ;

if $m \leq n$, the elements on and below the $(n-m)$ -th superdiagonal contain the m -by- n lower trapezoidal matrix L ; the remaining elements, with the array *tau*, represent the orthogonal/ unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

tau

(local).

REAL for *psgeql2*

DOUBLE PRECISION for *pdgeql2*

COMPLEX for `pcgeql2`
 COMPLEX*16 for `pzgeql2`.

Array, size $LOCc(ja+n-1)$. This array contains the scalar factors of the elementary reflectors. `tau` is tied to the distributed matrix `A`.

`work` On exit, `work(1)` returns the minimal and optimal `lwork`.

`info` (local). INTEGER.

If `info = 0`, the execution is successful. If `info < 0`: If the i -th argument is an array and the j -entry had an illegal value, then `info = - (i*100+j)`, if the i -th argument is a scalar and had an illegal value, then `info = -i`.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(ja+k-1) * \dots * H(ja+1) * H(ja)$, where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - \tau * v * v'$

where τ is a real/complex scalar, and v is a real/complex vector with $v(m-k+i+1: m) = 0$ and $v(m-k+i) = 1$; $v(1: m-k+i-1)$ is stored on exit in $A(ia:ia+m-k+i-2, ja+n-k+i-1)$, and τ in $TAU(ja+n-k+i-1)$.

p?geqr2

Computes a QR factorization of a general rectangular matrix (unblocked algorithm).

Syntax

Fortran:

```
call psgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgeqr2(m, n, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psgeqr2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdgeqr2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcgeqr2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );
void pzgeqr2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );
```

Include Files

- C: `mkl_scalapack.h`

Description

The `p?geqr2` routine computes a QR factorization of a real/complex distributed m -by- n matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q^*R$.

Input Parameters

<code>m</code>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<code>a</code>	(local). REAL for <code>psgeqr2</code> DOUBLE PRECISION for <code>pdgeqr2</code> COMPLEX for <code>pcgeqr2</code> COMPLEX*16 for <code>pzgeqr2</code> . Pointer into the local memory to an array of size ($lld_a, LOCc(ja+n-1)$). On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$ which is to be factored.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.
<code>desca</code>	(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix A .
<code>work</code>	(local). REAL for <code>psgeqr2</code> DOUBLE PRECISION for <code>pdgeqr2</code> COMPLEX for <code>pcgeqr2</code> COMPLEX*16 for <code>pzgeqr2</code> . This is a workspace array of size ($lwork$).
<code>lwork</code>	(local or global) INTEGER. The size of the array <code>work</code> . <code>lwork</code> is local input and must be at least $lwork \geq mp0 + \max(1, nq0)$, where $iwoff = \text{mod}(ia-1, mb_a)$, $icoff = \text{mod}(ja-1, nb_a)$, $iarow = \text{idxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$, $iacol = \text{idxg2p}(ja, nb_a, mycol, csrc_a, npcol)$, $mp0 = \text{numroc}(m+iwoff, mb_a, myrow, iarow, nprow)$, $nq0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npcol)$.

`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprov`, and `npcol` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pixerbla`.

Output Parameters

`a`

(local).

On exit, the elements on and above the diagonal of `sub(A)` contain the $\min(m,n)$ by n upper trapezoidal matrix R (R is upper triangular if $m \geq n$); the elements below the diagonal, with the array `tau`, represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see *Application Notes* below).

`tau`

(local).

REAL for `psgeqr2`

DOUBLE PRECISION for `pdgeqr2`

COMPLEX for `pcgeqr2`

COMPLEX*16 for `pzgeqr2`.

Array, size $LOCc(ja+\min(m,n)-1)$. This array contains the scalar factors of the elementary reflectors. `tau` is tied to the distributed matrix A .

`work`

On exit, `work(1)` returns the minimal and optimal `lwork`.

`info`

(local) INTEGER.

If `info = 0`, the execution is successful. If `info < 0`:

If the i -th argument is an array and the j -entry had an illegal value, then `info = - (i*100+j)`,

If the i -th argument is a scalar and had an illegal value, then `info = -i`.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$$Q = H(ja) * H(ja+1) * \dots * H(ja+k-1), \text{ where } k = \min(m, n).$$

Each $H(i)$ has the form

$$H(j) = I - tau * v * v',$$

where tau is a real/complex scalar, and v is a real/complex vector with $v(1:i-1) = 0$ and $v(i) = 1$; $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1)$, and tau in $TAU(ja+i-1)$.

p?gerq2

Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

Fortran:

```

call psgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pdgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pcgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)
call pzgerq2(m, n, a, ia, ja, desca, tau, work, lwork, info)

```

C:

```

void psgerq2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdgerq2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcgerq2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT
*info );
void pzgerq2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT
*info );

```

Include Files

- C: mkl_scalapack.h

Description

The p?gerq2 routine computes an *RQ* factorization of a real/complex distributed *m*-by-*n* matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = R*Q$.

Input Parameters*m*

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).

n

(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).

a

(local).

REAL for psgerq2

DOUBLE PRECISION for pdgerq2

COMPLEX for pcgerq2

COMPLEX*16 for pzgerq2.

Pointer into the local memory to an array of size $(\text{lld}_a, \text{LOCc}(ja+n-1))$.

On entry, this array contains the local pieces of the *m*-by-*n* distributed matrix $\text{sub}(A)$ which is to be factored.

ia, ja

(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix *A*, respectively.

desca

(global and local) INTEGER array, size (*dlen_*). The array descriptor for the distributed matrix *A*.

<i>work</i>	(local). REAL for psgerq2 DOUBLE PRECISION for pdgerq2 COMPLEX for pcgerq2 COMPLEX*16 for pzgerq2. This is a workspace array of size (<i>lwork</i>).
<i>lwork</i>	(local or global) INTEGER. The size of the array <i>work</i> . <i>lwork</i> is local input and must be at least $lwork \geq nq0 + \max(1, mp0)$, where $\begin{aligned} iroff &= \text{mod}(ia-1, mb_a), \quad icoff = \text{mod}(ja-1, nb_a), \\ iarow &= \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow), \\ iacol &= \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol), \quad mp0 = \\ &\text{numroc}(m+iroff, mb_a, myrow, iarow, nprow), \\ nq0 &= \text{numroc}(n+icoff, nb_a, mycol, iacol, npcol), \\ \text{indxg2p} \text{ and } \text{numroc} &\text{ are ScaLAPACK tool functions; } myrow, mycol, nprow, \\ \text{and } npcol &\text{ can be determined by calling the subroutine blacs_gridinfo.} \end{aligned}$ <p>If <i>lwork</i> = -1, then <i>lwork</i> is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.</p>
Output Parameters	
<i>a</i>	(local). On exit, if $m \leq n$, the upper triangle of $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the m -by- m upper triangular matrix R ; if $m \geq n$, the elements on and above the $(m-n)$ -th subdiagonal contain the m -by- n upper trapezoidal matrix R ; the remaining elements, with the array <i>tau</i> , represent the orthogonal/ unitary matrix Q as a product of elementary reflectors (see <i>Application Notes</i> below).
<i>tau</i>	(local). REAL for psgerq2 DOUBLE PRECISION for pdgerq2 COMPLEX for pcgerq2 COMPLEX*16 for pzgerq2. Array, size $LOCr(ia+m-1)$. This array contains the scalar factors of the elementary reflectors. <i>tau</i> is tied to the distributed matrix <i>A</i> .
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) INTEGER.

If $info = 0$, the execution is successful.

if $info < 0$: If the i -th argument is an array and the j -entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix Q is represented as a product of elementary reflectors

$Q = H(ia) * H(ia+1) * \dots * H(ia+k-1)$ for real flavors,

$Q = (H(ia))^H * (H(ia+1))^H * \dots * (H(ia+k-1))^H$ for complex flavors,

where $k = \min(m, n)$.

Each $H(i)$ has the form

$H(i) = I - tau * v * v'$,

where tau is a real/complex scalar, and v is a real/complex vector with $v(n-k+i+1:n) = 0$ and $v(n-k+i) = 1$; $v(1:n-k+i-1)$ for real flavors or $\text{conjg}(v(1:n-k+i-1))$ for complex flavors is stored on exit in $A(ia+m-k+i-1, ja:ja+n-k+i-2)$, and tau in $TAU(ia+m-k+i-1)$.

p?getf2

Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).

Syntax

Fortran:

```
call psgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pdgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pcgetf2(m, n, a, ia, ja, desca, ipiv, info)
call pzgetf2(m, n, a, ia, ja, desca, ipiv, info)
```

C:

```
void psgetf2 (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *ipiv , MKL_INT *info );
void pdgetf2 (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
void pcgetf2 (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
void pzgetf2 (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *ipiv , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?getf2 routine computes an LU factorization of a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ using partial pivoting with row interchanges.

The factorization has the form $\text{sub}(A) = P * L^* U$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and U is upper triangular (upper trapezoidal if $m < n$). This is the right-looking Parallel Level 2 BLAS version of the algorithm.

Input Parameters

m

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).

n

(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($nb_a - \text{mod}(ja-1, nb_a) \geq n \geq 0$).

a

(local).

REAL for psgetf2

DOUBLE PRECISION for pdgetf2

COMPLEX for pcgetf2

COMPLEX*16 for pzgetf2.

Pointer into the local memory to an array of size ($l1d_a, LOCr(ja+n-1)$).

On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$.

ia, ja

(global) INTEGER. The row and column indices in the global array *a* indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca

(global and local) INTEGER array, size (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

ipiv

(local) INTEGER.

Array, size ($LOCr(m_a) + mb_a$). This array contains the pivoting information. $ipiv(i) \rightarrow$ The global row that local row *i* was swapped with. This array is tied to the distributed matrix *A*.

info

(local). INTEGER.

If $info = 0$: successful exit.

If $info < 0$:

- if the *i*-th argument is an array and the *j*-entry had an illegal value, then $info = -(i*100+j)$,
- if the *i*-th argument is a scalar and had an illegal value, then $info = -i$.

If $info > 0$: If $info = k$, $u(ia+k-1, ja+k-1)$ is exactly zero. The factorization has been completed, but the factor *u* is exactly singular, and division by zero will occur if it is used to solve a system of equations.

p?labrd

Reduces the first nb rows and columns of a general rectangular matrix A to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A .

Syntax

Fortran:

```
call pslabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy,
jy, descy, work)

call pdlabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy,
jy, descy, work)

call pclabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy,
jy, descy, work)

call pzlabrd(m, n, nb, a, ia, ja, desca, d, e, tauq, taup, x, ix, jx, descx, y, iy,
jy, descy, work)
```

C:

```
void pslabrd (MKL_INT *m , MKL_INT *n , MKL_INT *nb , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *d , float *e , float *tauq , float *taup , float *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , float *work );

void pdlabrd (MKL_INT *m , MKL_INT *n , MKL_INT *nb , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *d , double *e , double *tauq , double *taup ,
double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *y , MKL_INT *iy ,
MKL_INT *jy , MKL_INT *descy , double *work );

void pclabrd (MKL_INT *m , MKL_INT *n , MKL_INT *nb , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *d , float *e , MKL_Complex8 *tauq , MKL_Complex8
*taup , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_Complex8
*y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_Complex8 *work );

void pzlabrd (MKL_INT *m , MKL_INT *n , MKL_INT *nb , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *d , double *e , MKL_Complex16 *tauq ,
MKL_Complex16 *taup , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_Complex16 *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_Complex16 *work );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?labrd` routine reduces the first nb rows and columns of a real/complex general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form by an orthogonal/unitary transformation $Q^* A * P$, and returns the matrices X and Y necessary to apply the transformation to the unreduced part of $\text{sub}(A)$.

If $m \geq n$, $\text{sub}(A)$ is reduced to upper bidiagonal form; if $m < n$, $\text{sub}(A)$ is reduced to lower bidiagonal form.

This is an auxiliary routine called by `p?gebrd`.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>nb</i>	(global) INTEGER. The number of leading rows and columns of $\text{sub}(A)$ to be reduced.
<i>a</i>	(local). REAL for pslabrd DOUBLE PRECISION for pdlabrd COMPLEX for pclabrd COMPLEX*16 for pzlabrd. Pointer into the local memory to an array of size(<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). On entry, this array contains the local pieces of the general distributed matrix $\text{sub}(A)$.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix A.
<i>ix</i> , <i>jx</i>	(global) INTEGER. The row and column indices in the global array <i>x</i> indicating the first row and the first column of the submatrix $\text{sub}(X)$, respectively.
<i>descx</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix X.
<i>iy</i> , <i>jy</i>	(global) INTEGER. The row and column indices in the global array <i>y</i> indicating the first row and the first column of the submatrix $\text{sub}(Y)$, respectively.
<i>descy</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix Y.
<i>work</i>	(local). REAL for pslabrd DOUBLE PRECISION for pdlabrd COMPLEX for pclabrd COMPLEX*16 for pzlabrd Workspace array, size (<i>lwork</i>) $lwork \geq nb_a + nq,$

```

with  $nq = \text{numroc}(n+\text{mod}(ia-1, nb\_y), nb\_y, mycol, iacol,$ 
 $npcol)$ 

 $iacol = \text{indxg2p}(ja, nb\_a, mycol, csrc\_a, npcol)$ 

 $\text{indxg2p}$  and  $\text{numroc}$  are ScaLAPACK tool functions;  $myrow$ ,  $mycol$ ,  $nprow$ , and  $npcol$  can be determined by calling the subroutine  $\text{blacs\_gridinfo}$ .

```

Output Parameters

- a* (local)
 On exit, the first nb rows and columns of the matrix are overwritten; the rest of the distributed matrix $\text{sub}(A)$ is unchanged.
 If $m \geq n$, elements on and below the diagonal in the first nb columns, with the array τ_{aq} , represent the orthogonal/unitary matrix Q as a product of elementary reflectors; and elements above the diagonal in the first nb rows, with the array τ_{ap} , represent the orthogonal/unitary matrix P as a product of elementary reflectors.
 If $m < n$, elements below the diagonal in the first nb columns, with the array τ_{aq} , represent the orthogonal/unitary matrix Q as a product of elementary reflectors, and elements on and above the diagonal in the first nb rows, with the array τ_{ap} , represent the orthogonal/unitary matrix P as a product of elementary reflectors. See *Application Notes* below.
- d* (local).
 REAL for pslabrd
 DOUBLE PRECISION for pdlabrd
 COMPLEX for pclabrd
 COMPLEX*16 for pzlabrd
 Array, size $\text{LOCr}(ia+\min(m,n)-1)$ if $m \geq n$; $\text{LOCr}(ja+\min(m,n)-1)$ otherwise. The distributed diagonal elements of the bidiagonal distributed matrix B :
 $d(i) = A(ia+i-1, ja+i-1).$
d is tied to the distributed matrix *A*.
- e* (local).
 REAL for pslabrd
 DOUBLE PRECISION for pdlabrd
 COMPLEX for pclabrd
 COMPLEX*16 for pzlabrd
 Array, size $\text{LOCr}(ia+\min(m,n)-1)$ if $m \geq n$; $\text{LOCr}(ja+\min(m,n)-2)$ otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix B :
 if $m \geq n$, $E(i) = A(ia+i-1, ja+i)$ for $i = 1, 2, \dots, n-1$;
 if $m < n$, $E(i) = A(ia+i, ja+i-1)$ for $i = 1, 2, \dots, m-1$.
e is tied to the distributed matrix *A*.

<i>tauq, taup</i>	(local).
	REAL for pslabrd
	DOUBLE PRECISION for pdlabrd
	COMPLEX for pclabrd
	COMPLEX*16 for pzlabrd
	Array size $LOC_c(ja+\min(m, n)-1)$ for <i>tauq</i> , size $LOC_r(ia+\min(m, n)-1)$ for <i>taup</i> . The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix <i>Q</i> for <i>tauq</i> , <i>P</i> for <i>taup</i> . <i>tauq</i> and <i>taup</i> are tied to the distributed matrix <i>A</i> . See <i>Application Notes</i> below.
<i>x</i>	(local)
	REAL for pslabrd
	DOUBLE PRECISION for pdlabrd
	COMPLEX for pclabrd
	COMPLEX*16 for pzlabrd
	Pointer into the local memory to an array of size lld_x by <i>nb</i> . On exit, the local pieces of the distributed <i>m</i> -by- <i>nb</i> matrix $X(ix:ix+m-1, jx:jx+nb-1)$ required to update the unreduced part of $\text{sub}(A)$.
<i>y</i>	(local).
	REAL for pslabrd
	DOUBLE PRECISION for pdlabrd
	COMPLEX for pclabrd
	COMPLEX*16 for pzlabrd
	Pointer into the local memory to an array of size lld_y by <i>nb</i> . On exit, the local pieces of the distributed <i>n</i> -by- <i>nb</i> matrix $Y(iy:iy+n-1, jy:jy+nb-1)$ required to update the unreduced part of $\text{sub}(A)$.

Application Notes

The matrices *Q* and *P* are represented as products of elementary reflectors:

$$Q = H(1) * H(2) * \dots * H(nb), \text{ and } P = G(1) * G(2) * \dots * G(nb)$$

Each *H(i)* and *G(i)* has the form:

$$H(i) = I - tauq * v * v', \text{ and } G(i) = I - taup * u * u',$$

where *tauq* and *taup* are real/complex scalars, and *v* and *u* are real/complex vectors.

If $m \geq n$, $v(1: i-1) = 0$, $v(i) = 1$, and $v(i:m)$ is stored on exit in

$A(ia+i-1:ia+m-1, ja+i-1); u(1:i) = 0, u(i+1) = 1$, and $u(i+1:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$; *tauq* is stored in *TAUQ(ja+i-1)* and *taup* in *TAUP(ia+i-1)*.

If $m < n$, $v(1: i) = 0$, $v(i+1) = 1$, and $v(i+1:m)$ is stored on exit in

$A(ia+i+1:ia+m-1, ja+i-1); u(1:i-1) = 0, u(i) = 1$, and $u(i:n)$ is stored on exit in $A(ia+i-1, ja+i:ja+n-1)$; *tauq* is stored in *TAUQ(ja+i-1)* and *taup* in *TAUP(ia+i-1)*. The elements of the vectors *v* and *u* together form the *m*-by-*nb* matrix *V* and the *nb*-by-*n* matrix *U'* which are necessary, with *X* and *Y*, to apply the transformation to the unreduced part of the matrix, using a block update of the form: $\text{sub}(A) := \text{sub}(A) - V * Y' - X * U'$. The contents of *sub(A)* on exit are illustrated by the following examples with *nb* = 2:

$$\begin{array}{ll}
 m = 6 \text{ and } n = 5(m > n): & m = 5 \text{ and } n = 6(m < n) : \\
 \left[\begin{array}{cccc} 1 & 1 & u_1 & v_1 \\ v_1 & 1 & 1 & u_2 \\ v_1 & v_2 & a & a \end{array} \right] & \left[\begin{array}{cccccc} 1 & u_1 & v_1 & u_1 & v_1 \\ 1 & 1 & u_2 & u_2 & u_2 \\ v_1 & 1 & a & a & a \\ v_1 & v_2 & a & a & a \\ v_1 & v_2 & a & a & a \end{array} \right]
 \end{array}$$

where a denotes an element of the original matrix which is unchanged, v_i denotes an element of the vector defining $H(i)$, and u_i an element of the vector defining $G(i)$.

p?lacon

Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.

Syntax

Fortran:

```
call psblacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pdblacon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pclaon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
call pzlaon(n, v, iv, jv, descv, x, ix, jx, descx, isgn, est, kase)
```

C:

```
void psblaon (MKL_INT *n , float *v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv ,
float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *isgn , float *est ,
MKL_INT *kase );
void pdblaon (MKL_INT *n , double *v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv ,
double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *isgn , double *est ,
MKL_INT *kase );
void pclaon (MKL_INT *n , MKL_Complex8 *v , MKL_INT *iv , MKL_INT *jv , MKL_INT
*descv , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *est ,
MKL_INT *kase );
void pzlaon (MKL_INT *n , MKL_Complex16 *v , MKL_INT *iv , MKL_INT *jv , MKL_INT
*descv , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *est ,
MKL_INT *kase );
```

Include Files

- C: mkl_scalapack.h

Description

The p?lacon routine estimates the 1-norm of a square, real/unitary distributed matrix A. Reverse communication is used for evaluating matrix-vector products. x and v are aligned with the distributed matrix A, this information is implicitly contained within iv, ix, descv, and descx.

Input Parameters

n (global) INTEGER. The length of the distributed vectors v and x. $n \geq 0$.

<i>v</i>	(local). REAL for pslacon DOUBLE PRECISION for pdlacon COMPLEX for pclacon COMPLEX*16 for pzlacon. Pointer into the local memory to an array of size $LOCr(n+\text{mod}(iv-1, mb_v))$. On the final return, $v = a^*w$, where $\text{est} = \text{norm}(v)/\text{norm}(w)$ (w is not returned).
<i>iv, jv</i>	(global) INTEGER. The row and column indices in the global array <i>v</i> indicating the first row and the first column of the submatrix <i>V</i> , respectively.
<i>descv</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>V</i> .
<i>x</i>	(local). REAL for pslacon DOUBLE PRECISION for pdlacon COMPLEX for pclacon COMPLEX*16 for pzlacon. Pointer into the local memory to an array of size $LOCr(n+\text{mod}(ix-1, mb_x))$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the global array <i>x</i> indicating the first row and the first column of the submatrix <i>X</i> , respectively.
<i>descx</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>X</i> .
<i>isgn</i>	(local). INTEGER. Array, size $LOCr(n+\text{mod}(ix-1, mb_x))$. <i>isgn</i> is aligned with <i>x</i> and <i>v</i> .
<i>kase</i>	(local). INTEGER. On the initial call to p?lacon, <i>kase</i> should be 0.

Output Parameters

<i>x</i>	(local). On an intermediate return, <i>X</i> should be overwritten by A^*X , if <i>kase</i> =1, A^*X , if <i>kase</i> =2, p?lacon must be re-called with all the other parameters unchanged.
<i>est</i>	(global). REAL for single precision flavors DOUBLE PRECISION for double precision flavors
<i>kase</i>	(local)

INTEGER. On an intermediate return, *kase* is 1 or 2, indicating whether *X* should be overwritten by A^*X , or A'^*X . On the final return from p?lacon, *kase* is again 0.

p?laconsb

Looks for two consecutive small subdiagonal elements.

Syntax

Fortran:

```
call psblaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
call pdblaconsb(a, desca, i, l, m, h44, h33, h43h34, buf, lwork)
```

C:

```
void psblaconsb (float *a , MKL_INT *desca , MKL_INT *i , MKL_INT *l , MKL_INT *m ,
float *h44 , float *h33 , float *h43h34 , float *buf , MKL_INT *lwork );
void pdblaconsb (double *a , MKL_INT *desca , MKL_INT *i , MKL_INT *l , MKL_INT *m ,
double *h44 , double *h33 , double *h43h34 , double *buf , MKL_INT *lwork );
```

Include Files

- C: mkl_scalapack.h

Description

The p?laconsb routine looks for two consecutive small subdiagonal elements by analyzing the effect of starting a double shift QR iteration given by *h44*, *h33*, and *h43h34* to see if this process makes a subdiagonal negligible.

Input Parameters

<i>a</i>	(global). REAL for psblaconsb DOUBLE PRECISION for pdblaconsb Array, size (<i>desca</i> (lld_),*). On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.
<i>desca</i>	(global and local) INTEGER. Array of size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>i</i>	(global) INTEGER. The global location of the bottom of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>l</i>	(global) INTEGER. The global location of the top of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>h44</i> , <i>h33</i> , <i>h43h34</i>	(global). REAL for psblaconsb DOUBLE PRECISION for pdblaconsb These three values are for the double shift QR iteration.

lwork (local or global) INTEGER.
 This must be at least $7 * \text{ceil}(\text{ceil}((i-1)/\text{hbl})/\text{lcm}(\text{nproc}, \text{npcol}))$.
 Here lcm is least common multiple and $\text{nproc} * \text{npcol}$ is the logical grid size.

Output Parameters

m (global). On exit, this yields the starting location of the QR double shift.
 This will satisfy:
 $i \leq m \leq i-2$.

buf (local).
 REAL for pslaconsb
 DOUBLE PRECISION for pdlaconsb
 Array of size *lwork*.

lwork (local or global). On exit, *lwork* is the size of the work buffer.

p?lacp2

Copies all or part of a distributed matrix to another distributed matrix.

Syntax

Fortran:

```
call pslacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pdlacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pclacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pzlacp2(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

C:

```
void pslacp2 (char *uplo , MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );
void pdlacp2 (char *uplo , MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );
void pclacp2 (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb );
void pzlacp2 (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?lacp2` routine copies all or part of a distributed matrix A to another distributed matrix B . No communication is performed, `p?lacp2` performs a local copy $\text{sub}(A) := \text{sub}(B)$, where $\text{sub}(A)$ denotes $A(ia:ia+m-1, a:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+m-1, jb:jb+n-1)$.

`p?lacp2` requires that only dimension of the matrix operands is distributed.

Input Parameters

<code>uplo</code>	(global) CHARACTER. Specifies the part of the distributed matrix $\text{sub}(A)$ to be copied: = 'U': Upper triangular part is copied; the strictly lower triangular part of $\text{sub}(A)$ is not referenced; = 'L': Lower triangular part is copied; the strictly upper triangular part of $\text{sub}(A)$ is not referenced. Otherwise: all of the matrix $\text{sub}(A)$ is copied.
<code>m</code>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<code>a</code>	(local). REAL for <code>pslacp2</code> DOUBLE PRECISION for <code>pdlacp2</code> COMPLEX for <code>pclacp2</code> COMPLEX*16 for <code>pzlacp2</code> . Pointer into the local memory to an array of size (<code>lld_a, LOCc(ja+n-1)</code>). On entry, this array contains the local pieces of the m -by- n distributed matrix $\text{sub}(A)$.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of $\text{sub}(A)$, respectively.
<code>desca</code>	(global and local) INTEGER array, size (<code>dlen_</code>). The array descriptor for the distributed matrix A .
<code>ib, jb</code>	(global) INTEGER. The row and column indices in the global array B indicating the first row and the first column of $\text{sub}(B)$, respectively.
<code>descb</code>	(global and local) INTEGER array, size (<code>dlen_</code>). The array descriptor for the distributed matrix B .

Output Parameters

<code>b</code>	(local). REAL for <code>pslacp2</code>
----------------	---

DOUBLE PRECISION for pdlacp2

COMPLEX for pclacp2

COMPLEX*16 for pzlacp2.

Pointer into the local memory to an array of size (lld_b , $LOCc(jb+n-1)$). This array contains on exit the local pieces of the distributed matrix sub(B) set as follows:

```
if uplo = 'U',  $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$ ,  $1 \leq i \leq j$ ,  $1 \leq j \leq n$ ;
if uplo = 'L',  $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$ ,  $j \leq i \leq m$ ,  $1 \leq j \leq n$ ;
otherwise,  $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$ ,  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ .
```

p?lacp3

Copies from a global parallel array into a local replicated array or vice versa.

Syntax

Fortran:

```
call pslacp3(m, i, a, desca, b, ldb, ii, jj, rev)
call pdlacp3(m, i, a, desca, b, ldb, ii, jj, rev)
```

C:

```
void pslacp3 (MKL_INT *m , MKL_INT *i , float *a , MKL_INT *desca , float *b , MKL_INT
*ldb , MKL_INT *ii , MKL_INT *jj , MKL_INT *rev );
void pdlacp3 (MKL_INT *m , MKL_INT *i , double *a , MKL_INT *desca , double *b ,
MKL_INT *ldb , MKL_INT *ii , MKL_INT *jj , MKL_INT *rev );
```

Include Files

- C: mkl_scalapack.h

Description

This is an auxiliary routine that copies from a global parallel array into a local replicated array or vice versa. Note that the entire submatrix that is copied gets placed on one node or more. The receiving node can be specified precisely, or all nodes can receive, or just one row or column of nodes.

Input Parameters

m

(global) INTEGER.

m is the order of the square submatrix that is copied.

m ≥ 0 . Unchanged on exit.

i

(global) INTEGER. $A(i, i)$ is the global location that the copying starts from. Unchanged on exit.

a

(global). REAL for pslacp3

DOUBLE PRECISION for pdlacp3

Array, size ($desca(lld_), *$). On entry, the parallel matrix to be copied into or from.

desca (global and local) INTEGER array, size (*dlen_*). The array descriptor for the distributed matrix *A*.

b (local).
REAL for pslacp3
DOUBLE PRECISION for pdlacp3
Array, size *ldb* by *m*. If *rev* = 0, this is the global portion of the array *A*(*i*:*i+m-1*, *i*:*i+m-1*). If *rev* = 1, this is the unchanged on exit.

ldb (local)
INTEGER.
The leading dimension of *B*.

ii (global) INTEGER. By using *rev* 0 and 1, data can be sent out and returned again. If *rev* = 0, then *ii* is destination row index for the node(s) receiving the replicated *B*. If *ii* ≥ 0, *jj* ≥ 0, then node (*ii*, *jj*) receives the data. If *ii* = -1, *jj* ≥ 0, then all rows in column *jj* receive the data. If *ii* ≥ 0, *jj* = -1, then all cols in row *ii* receive the data. If *ii* = -1, *jj* = -1, then all nodes receive the data. If *rev* != 0, then *ii* is the source row index for the node(s) sending the replicated *B*.

jj (global) INTEGER. Similar description as *ii* above.

rev (global) INTEGER. Use *rev* = 0 to send global *A* into locally replicated *B* (on node (*ii*, *jj*)). Use *rev* != 0 to send locally replicated *B* from node (*ii*, *jj*) to its owner (which changes depending on its location in *A*) into the global *A*.

Output Parameters

a (global). On exit, if *rev* = 1, the copied data. Unchanged on exit if *rev* = 0.

b (local). If *rev* = 1, this is unchanged on exit.

p?lacpy

Copies all or part of one two-dimensional array to another.

Syntax

Fortran:

```
call pslacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pdlacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pclacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
call pzlacpy(uplo, m, n, a, ia, ja, desca, b, ib, jb, descb)
```

C:

```
void pslacpy (char *uplo , MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );
```

```

void pdlacpy (char *uplo , MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );
void pclacpy (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb );
void pzlacpy (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb );

```

Include Files

- C: mkl_scalapack.h

Description

The p?lacpy routine copies all or part of a distributed matrix A to another distributed matrix B . No communication is performed, p?lacpy performs a local copy $\text{sub}(B) := \text{sub}(A)$, where $\text{sub}(A)$ denotes $A(ia:ia+m-1, ja:ja+n-1)$ and $\text{sub}(B)$ denotes $B(ib:ib+m-1, jb:jb+n-1)$.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Specifies the part of the distributed matrix $\text{sub}(A)$ to be copied:
	= 'U': Upper triangular part; the strictly lower triangular part of $\text{sub}(A)$ is not referenced;
	= 'L': Lower triangular part; the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
	Otherwise: all of the matrix $\text{sub}(A)$ is copied.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). REAL for pslacpy DOUBLE PRECISION for pdlacpy COMPLEX for pclacpy COMPLEX*16 for pzlacpy. Pointer into the local memory to an array of size (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). On entry, this array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array <i>a</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

<i>desca</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix A.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the global array <i>B</i> indicating the first row and the first column of sub(<i>B</i>) respectively.
<i>descb</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix A.

Output Parameters

<i>b</i>	(local). REAL for pslacpy DOUBLE PRECISION for pdlacpy COMPLEX for pclacpy COMPLEX*16 for pzlacpy. Pointer into the local memory to an array of size (<i>lld_b, LOCc(jb+n-1)</i>). This array contains on exit the local pieces of the distributed matrix sub(B) set as follows: if <i>uplo</i> = 'U', <i>B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1), 1 ≤ i ≤ j, 1 ≤ j ≤ n</i> ; if <i>uplo</i> = 'L', <i>B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1), j ≤ i ≤ m, 1 ≤ j ≤ n</i> ; otherwise, <i>B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1), 1 ≤ i ≤ m, 1 ≤ j ≤ n</i> .
----------	--

p?laevswp

Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.

Syntax

Fortran:

```
call pslaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pdlaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pclaevevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
call pzlaevswp(n, zin, ldzi, z, iz, jz, descz, nvs, key, rwork, lrwork)
```

C:

```
void pslaevswp (MKL_INT *n , float *zin , MKL_INT *ldzi , float *z , MKL_INT *iz ,
MKL_INT *jz , MKL_INT *descz , MKL_INT *nvs , MKL_INT *key , float *work , MKL_INT
*lwrook );
void pdlaevswp (MKL_INT *n , double *zin , MKL_INT *ldzi , double *z , MKL_INT *iz ,
MKL_INT *jz , MKL_INT *descz , MKL_INT *nvs , MKL_INT *key , double *work , MKL_INT
*lwrook );
void pclaevevswp (MKL_INT *n , float *zin , MKL_INT *ldzi , MKL_Complex8 *z , MKL_INT
*iz , MKL_INT *jz , MKL_INT *descz , MKL_INT *nvs , MKL_INT *key , float *rwork ,
MKL_INT *lrwork );
```

```
void pzlaevswp (MKL_INT *n , double *zin , MKL_INT *ldzi , MKL_Complex16 *z , MKL_INT
*iz , MKL_INT *jz , MKL_INT *descz , MKL_INT *nvs , MKL_INT *key , double *rwork ,
MKL_INT *lrwork );
```

Include Files

- C: mkl_scalapack.h

Description

The p?laevswp routine moves the eigenvectors (potentially unsorted) from where they are computed, to a ScaLAPACK standard block cyclic array, sorted so that the corresponding eigenvalues are sorted.

Input Parameters

np = the number of rows local to a given process.

nq = the number of columns local to a given process.

n (global) INTEGER.

The order of the matrix *A*. $n \geq 0$.

zin (local).

REAL for pslaevswp

DOUBLE PRECISION for pdlaevswp

COMPLEX for pclaevewp

COMPLEX*16 for pzlaevswp. Array, size (*ldzi*, *nvs(iam)*). The eigenvectors on input. Each eigenvector resides entirely in one process. Each process holds a contiguous set of *nvs(iam)* eigenvectors. The first eigenvector which the process holds is: sum for $i=[0, iam-1]$ of *nvs(i)*.

ldzi (local)

INTEGER. The leading dimension of the *zin* array.

iz, jz (global) INTEGER. The row and column indices in the global array *Z* indicating the first row and the first column of the submatrix *Z*, respectively.

descz (global and local) INTEGER array, size (*dlen_*). The array descriptor for the distributed matrix *Z*.

nvs (global) INTEGER.

Array, size (*nprocs*+1)

nvs(i) = number of processes number of eigenvectors held by processes [0, *i*-1)

nvs(1) = number of eigen vectors held by[0, 1 -1] = 0

nvs(nprocs+1)= number of eigen vectors held by [0, *nprocs*)= total number of eigenvectors.

key (global) INTEGER.

Array, size (*n*). Indicates the actual index (after sorting) for each of the eigenvectors.

rwork (local).
 REAL for pslaevswp
 DOUBLE PRECISION for pdlaevswp
 COMPLEX for pclaevswp
 COMPLEX*16 for pzlaevswp. Array, size (*lrwork*).

lrwork (local)
 INTEGER. Size of *work*.

Output Parameters

z (local).
 REAL for pslaevswp
 DOUBLE PRECISION for pdlaevswp
 COMPLEX for pclaevswp
 COMPLEX*16 for pzlaevswp.
 Array, global size *n* by *n*, local size (*descz(dlen_)*, *nq*). The eigenvectors on output. The eigenvectors are distributed in a block cyclic manner in both dimensions, with a block size of *nb*.

p?lahrd

Reduces the first nb columns of a general rectangular matrix A so that elements below the k-th subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of A.

Syntax

Fortran:

```
call pslahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pdlahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pclahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
call pzlahrd(n, k, nb, a, ia, ja, desca, tau, t, y, iy, jy, descy, work)
```

C:

```
void pslahrd (MKL_INT *n , MKL_INT *k , MKL_INT *nb , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *t , float *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , float *work );
void pdlahrd (MKL_INT *n , MKL_INT *k , MKL_INT *nb , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *tau , double *t , double *y , MKL_INT *iy ,
MKL_INT *jy , MKL_INT *descy , double *work );
void pclahrd (MKL_INT *n , MKL_INT *k , MKL_INT *nb , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *t , MKL_Complex8 *y ,
MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_Complex8 *work );
```

```
void pzlahrd (MKL_INT *n , MKL_INT *k , MKL_INT *nb , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *t , MKL_Complex16
*y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_Complex16 *work );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?lahrd` routine reduces the first nb columns of a real general n -by- $(n-k+1)$ distributed matrix $A(ia:ia+n-1, ja:ja+n-k)$ so that elements below the k -th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation Q^*A^*Q . The routine returns the matrices V and T which determine Q as a block reflector $I-V^*T^*V'$, and also the matrix $Y = A^*V^*T$.

This is an auxiliary routine called by `p?gehrd`. In the following comments `sub(A)` denotes $A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

<code>n</code>	(global) INTEGER.
	The order of the distributed submatrix <code>sub(A)</code> . $n \geq 0$.
<code>k</code>	(global) INTEGER.
	The offset for the reduction. Elements below the k -th subdiagonal in the first nb columns are reduced to zero.
<code>nb</code>	(global) INTEGER.
	The number of columns to be reduced.
<code>a</code>	(local). REAL for <code>pslahrd</code> DOUBLE PRECISION for <code>pdlahrd</code> COMPLEX for <code>pclahrd</code> COMPLEX*16 for <code>pzlahrd</code> . Pointer into the local memory to an array of size (<code>lld_a</code> , <code>LOCc(ja+n-k)</code>). On entry, this array contains the local pieces of the n -by- $(n-k+1)$ general distributed matrix $A(ia:ia+n-1, ja:ja+n-k)$.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<code>desca</code>	(global and local) INTEGER array, size (<code>dlen_</code>). The array descriptor for the distributed matrix A .
<code>iy, jy</code>	(global) INTEGER. The row and column indices in the global array Y indicating the first row and the first column of the submatrix <code>sub(Y)</code> , respectively.
<code>descy</code>	(global and local) INTEGER array, size (<code>dlen_</code>). The array descriptor for the distributed matrix Y .
<code>work</code>	(local).

```

REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Array, size (nb).

```

Output Parameters

a (local).
On exit, the elements on and above the k -th subdiagonal in the first nb columns are overwritten with the corresponding elements of the reduced distributed matrix; the elements below the k -th subdiagonal, with the array *tau*, represent the matrix Q as a product of elementary reflectors. The other columns of $A(ia:ia+n-1, ja:ja+n-k)$ are unchanged. (See *Application Notes* below.)

tau (local)
REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Array, size $LOC(ja+n-2)$. The scalar factors of the elementary reflectors (see *Application Notes* below). *tau* is tied to the distributed matrix *A*.

t (local) REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Array, size nb_a by nb_a The upper triangular matrix T .

y (local).
REAL for pslahrd
DOUBLE PRECISION for pdlahrd
COMPLEX for pclahrd
COMPLEX*16 for pzlahrd.
Pointer into the local memory to an array of size lId_y by nb_a . On exit, this array contains the local pieces of the n -by- nb distributed matrix Y . $lId_y \geq LOC(ia+n-1)$.

Application Notes

The matrix Q is represented as a product of nb elementary reflectors

$$Q = H(1)*H(2)*\dots*H(nb).$$

Each $H(i)$ has the form

$$H(i) = i - tau * v * v^T,$$

where τ_{ii} is a real/complex scalar, and v is a real/complex vector with $v(1:i+k-1) = 0$, $v(i+k) = 1$; $v(i+k+1:n)$ is stored on exit in $A(ia+i+k:ia+n-1, ja+i-1)$, and τ_{ii} in $TAU(ja+i-1)$.

The elements of the vectors v together form the $(n-k+1)$ -by- nb matrix V which is needed, with T and Y , to apply the transformation to the unreduced part of the matrix, using an update of the form: $A(ia:ia+n-1, ja:ja+n-k) := (I - V^*T^*V') * (A(ia:ia+n-1, ja:ja+n-k) - Y^*V')$. The contents of $A(ia:ia+n-1, ja:ja+n-k)$ on exit are illustrated by the following example with $n = 7$, $k = 3$, and $nb = 2$:

$$\begin{bmatrix} a & h & a & a & a \\ a & h & a & a & a \\ a & h & a & a & a \\ h & h & a & a & a \\ v1 & h & a & a & a \\ v1 & v2 & a & a & a \\ v1 & v2 & a & a & a \end{bmatrix}$$

where a denotes an element of the original matrix $A(ia:ia+n-1, ja:ja+n-k)$, h denotes a modified element of the upper Hessenberg matrix H , and vi denotes an element of the vector defining $H(i)$.

p?laiect

Exploits IEEE arithmetic to accelerate the computations of eigenvalues. (C interface function).

Syntax

Fortran:

```
void pslaiect(float *sigma, int *n, float *d, int *count);
void pdlaiectb(float *sigma, int *n, float *d, int *count);
void pdlaiectl(float *sigma, int *n, float *d, int *count);
```

C:

```
void pslaiect (float *sigma , MKL_INT *n , float *d , MKL_INT *count );
void pdlaiectb (float *sigma , MKL_INT *n , float *d , MKL_INT *count );
void pdlaiectl (float *sigma , MKL_INT *n , float *d , MKL_INT *count );
```

Include Files

- C: mkl_scalapack.h

Description

The p?laiect routine computes the number of negative eigenvalues of $(A - \sigma I)$. This implementation of the Sturm Sequence loop exploits IEEE arithmetic and has no conditionals in the innermost loop. The signbit for real routine pslaiect is assumed to be bit 32. Double precision routines pdlaiectb and pdlaiectl differ in the order of the double precision word storage and, consequently, in the signbit location. For pdlaiectb, the double precision word is stored in the big-endian word order and the signbit is assumed to be bit 32. For pdlaiectl, the double precision word is stored in the little-endian word order and the signbit is assumed to be bit 64.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code.

This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

<i>sigma</i>	REAL for pslaiect DOUBLE PRECISION for pdlaiectb/pdlaiectl. The shift. <i>p?laiect</i> finds the number of eigenvalues less than equal to <i>sigma</i> .
<i>n</i>	INTEGER. The order of the tridiagonal matrix <i>T</i> . $n \geq 1$.
<i>d</i>	REAL for pslaiect DOUBLE PRECISION for pdlaiectb/pdlaiectl. Array of size $(2n - 1)$. On entry, this array contains the diagonals and the squares of the off-diagonal elements of the tridiagonal matrix <i>T</i> . These elements are assumed to be interleaved in memory for better cache performance. The diagonal entries of <i>T</i> are in the entries $d(1), d(3), \dots, d(2n-1)$, while the squares of the off-diagonal entries are $d(2), d(4), \dots, d(2n-2)$. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than $overflow^{(1/2)} * underflow^{(1/4)}$ in absolute value, and for greatest accuracy, it should not be much smaller than that.

Output Parameters

<i>n</i>	INTEGER. The count of the number of eigenvalues of <i>T</i> less than or equal to <i>sigma</i> .
----------	--

p?lamve

Copies all or part of one two-dimensional distributed array to another.

Syntax

Fortran:

```
call pslamve( uplo, m, n, a, ia, ja, desca, b, ib, jb, descb, dwork )
call pdlamve( uplo, m, n, a, ia, ja, desca, b, ib, jb, descb, dwork )
```

Include Files

- C: mkl_scalapack.h

Description

p?lamve copies all or part of a distributed matrix *A* to another distributed matrix *B*. There is no alignment assumptions at all except that *A* and *B* are of the same size.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1 Specifies the part of the distributed matrix sub(<i>A</i>) to be copied: = 'U': Upper triangular part is copied; the strictly lower triangular part of sub(<i>A</i>) is not referenced;
-------------	--

= 'L': Lower triangular part is copied; the strictly upper triangular part of sub(A) is not referenced;

Otherwise: All of the matrix sub(A) is copied.

m

(global) INTEGER

The number of rows to be operated on, which is the number of rows of the distributed submatrix sub(A). $m \geq 0$.

n

(global) INTEGER

The number of columns to be operated on, which is the number of columns of the distributed submatrix sub(A). $n \geq 0$.

a

REAL for pslamve

DOUBLE PRECISION for pdlamve

(local) pointer into the local memory to an array of dimension (lld_a , $\text{LOC}_c(j_a+n-1)$). This array contains the local pieces of the distributed matrix sub(A) to be copied from.

i_a

(global) INTEGER

The row index in the global array a indicating the first row of sub(A).

j_a

(global) INTEGER

The column index in the global array a indicating the first column of sub(A).

desc_a

(global and local) INTEGER array of dimension $dlen_$.

The array descriptor for the distributed matrix A .

i_b

(global) INTEGER

The row index in the global array b indicating the first row of sub(B).

j_b

(global) INTEGER

The column index in the global array b indicating the first column of sub(B).

desc_b

(global and local) INTEGER array of dimension $dlen_$.

The array descriptor for the distributed matrix B .

$dwork$

REAL for pslamve

DOUBLE PRECISION for pdlamve

(local workspace) array

If $\text{uplo} = 'U'$ or $\text{uplo} = 'L'$ and number of processors > 1, the length of $dwork$ is at least as large as the length of b .

Otherwise, $dwork$ is not referenced.

OUTPUT Parameters

b

REAL for pslamve

DOUBLE PRECISION for pdlamve
 (local) pointer into the local memory to an array of dimension (lld_b ,
 $LOC_c(jb+n-1)$). This array contains on exit the local pieces of the
 distributed matrix sub(B).

p?lange

Returns the value of the 1-norm, Frobenius norm,
 infinity-norm, or the largest absolute value of any
 element, of a general rectangular matrix.

Syntax

Fortran:

```
val = pslange(norm, m, n, a, ia, ja, desca, work)
val = pdlange(norm, m, n, a, ia, ja, desca, work)
val = pclange(norm, m, n, a, ia, ja, desca, work)
val = pzlange(norm, m, n, a, ia, ja, desca, work)
```

C:

```
float pslange (char *norm , MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *work );
double pdlange (char *norm , MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *work );
float pclange (char *norm , MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *work );
double pzlange (char *norm , MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *work );
```

Include Files

- C: mkl_scalapack.h

Description

The p?lange routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix sub(A) = $A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

norm (global) CHARACTER. Specifies what value is returned by the routine:
 = 'M' or 'm': val = max(abs(A_{ij})), largest absolute value of the matrix A , it's not a matrix norm.
 = '1' or 'O' or 'o': val = norm1(A), 1-norm of the matrix A (maximum column sum),
 = 'I' or 'i': val = normI(A), infinity norm of the matrix A (maximum row sum),
 = 'F', 'f', 'E' or 'e': val = normF(A), Frobenius norm of the matrix A (square root of sum of squares).

m (global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix sub(*A*). When *m* = 0, p?lange is set to zero. *m* ≥ 0.

n (global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). When *n* = 0, p?lange is set to zero. *n* ≥ 0.

a (local).

REAL for pslange

DOUBLE PRECISION for pdlange

COMPLEX for pclange

COMPLEX*16 for pzrange.

Pointer into the local memory to an array of size (*lld_a*, LOCc(*ja*+*n*-1)) containing the local pieces of the distributed matrix sub(*A*).

ia, *ja* (global) INTEGER. The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca (global and local) INTEGER array, size (*dlen_*). The array descriptor for the distributed matrix *A*.

work (local).

REAL for pslange

DOUBLE PRECISION for pdlange

COMPLEX for pclange

COMPLEX*16 for pzrange.

Array size (*lwork*).

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),

nq0 if *norm* = '1', 'O' or 'o',

mp0 if *norm* = 'I' or 'i',

0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),

where

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nprov*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *npcol*),

mp0 = numroc(*m+iroffa*, *mb_a*, *myrow*, *iarow*, *nprov*),

nq0 = numroc(*n+icoffa*, *nb_a*, *mycol*, *iacol*, *npcol*),

indxg2p and numroc are ScaLAPACK tool routines; *myrow*, *mycol*, *nprov*, and *npcol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

`val`

REAL for `pslange/pclange`
DOUBLE PRECISION for `pdlange/pzlanh`
The value returned by the routine.

p?lanhs

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.

Syntax

Fortran:

```
val = pslanh(n, n, a, ia, ja, desca, work)
val = pdlanh(n, n, a, ia, ja, desca, work)
val = pclanh(n, n, a, ia, ja, desca, work)
val = pzlanh(n, n, a, ia, ja, desca, work)
```

C:

```
float pslanh (char *norm , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *work );
double pdlanh (char *norm , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *work );
float pclanh (char *norm , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *work );
double pzlanh (char *norm , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *work );
```

Include Files

- C: `mkl_scalapack.h`

Description

The `p?lanhs` routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an upper Hessenberg distributed matrix `sub(A) = A(ia:ia+m-1, ja:ja+n-1)`.

Input Parameters

`norm`

CHARACTER*1. Specifies the value to be returned by the routine:
= 'M' or 'm': `val = max(abs(Aij))`, largest absolute value of the matrix A .
= '1' or 'O' or 'o': `val = norm1(A)`, 1-norm of the matrix A (maximum column sum),
= 'I' or 'i': `val = normI(A)`, infinity norm of the matrix A (maximum row sum),

= 'F', 'f', 'E' or 'e': *val* = normF(*A*), Frobenius norm of the matrix *A* (square root of sum of squares).

n

(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix sub(*A*). When *n* = 0, p?lanhs is set to zero. *n* ≥ 0 .

a

(local).

REAL for pslanh

DOUBLE PRECISION for pdlanh

COMPLEX for pclanh

COMPLEX*16 for pzlanh.

Pointer into the local memory to an array of size (*lld_a*, *LOCc(ja+n-1)*) containing the local pieces of the distributed matrix sub(*A*).

ia, *ja*

(global) INTEGER.

The row and column indices in the global array *A* indicating the first row and the first column of the submatrix sub(*A*), respectively.

desca

(global and local) INTEGER array, size (*dlen_*). The array descriptor for the distributed matrix *A*.

work

(local).

REAL for pslanh

DOUBLE PRECISION for pdlanh

COMPLEX for pclanh

COMPLEX*16 for pzlanh.

Array, size (*lwork*).

lwork ≥ 0 if *norm* = 'M' or 'm' (not referenced),

nq0 if *norm* = '1', 'O' or 'o',

mp0 if *norm* = 'I' or 'i',

0 if *norm* = 'F', 'f', 'E' or 'e' (not referenced),

where

iroffa = mod(*ia*-1, *mb_a*), *icoffa* = mod(*ja*-1, *nb_a*),

iarow = indxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nrow*),

iacol = indxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *ncol*),

mp0 = numroc(*m+iroffa*, *mb_a*, *myrow*, *iarow*, *nrow*),

nq0 = numroc(*n+icoffa*, *nb_a*, *mycol*, *iacol*, *ncol*),

indxg2p and numroc are ScaLAPACK tool routines; *myrow*, *imycol*, *nrow*, and *ncol* can be determined by calling the subroutine blacs_gridinfo.

Output Parameters

`val` The value returned by the function.

p?lansy, p?lanhe

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a real symmetric or a complex Hermitian matrix.

Syntax

Fortran:

```
val = pslansy(norm, uplo, n, a, ia, ja, desca, work)
val = pdlansy(norm, uplo, n, a, ia, ja, desca, work)
val = pclansy(norm, uplo, n, a, ia, ja, desca, work)
val = pzlanhsy(norm, uplo, n, a, ia, ja, desca, work)
val = pclanhe(norm, uplo, n, a, ia, ja, desca, work)
val = pzlanhe(norm, uplo, n, a, ia, ja, desca, work)
```

C:

```
float pslansy (char *norm , char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *work );
double pdlansy (char *norm , char *uplo , MKL_INT *n , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *work );
float pclansy (char *norm , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *work );
double pzlanhsy (char *norm , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *work );
float pclanhe (char *norm , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *work );
double pzlanhe (char *norm , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *work );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?lansy` and `p?lanhe` routines return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

`norm` (global) CHARACTER. Specifies what value is returned by the routine:
 $= 'M'$ or '`m`': $val = \max(|A_{ij}|)$, largest absolute value of the matrix A , it is not a matrix norm.

= '1' or 'O' or 'o': $val = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum),
= 'I' or 'i': $val = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum),
= 'F', 'f', 'E' or 'e': $val = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).

uplo

(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric matrix $\text{sub}(A)$ is to be referenced.

= 'U': Upper triangular part of $\text{sub}(A)$ is referenced,
= 'L': Lower triangular part of $\text{sub}(A)$ is referenced.

n

(global) INTEGER.

The number of columns to be operated on i.e the number of columns of the distributed submatrix $\text{sub}(A)$. When $n = 0$, $p?lansy$ is set to zero. $n \geq 0$.

a

(local).

```
REAL for pslansy
DOUBLE PRECISION for pdlansy
COMPLEX for pclansy, pclanhe
COMPLEX*16 for pzlanhs, pzlanhe.
```

Pointer into the local memory to an array of size ($lld_a, LOCc(ja+n-1)$) containing the local pieces of the distributed matrix $\text{sub}(A)$.

If $uplo = 'U'$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular matrix whose norm is to be computed, and the strictly lower triangular part of this matrix is not referenced. If $uplo = 'L'$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular matrix whose norm is to be computed, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

ia, ja

(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca

(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix A .

work

(local).

```
REAL for pslansy, pclansy, pclanhe
DOUBLE PRECISION for pdlansy, pzlanhs, pzlanhe
```

Array size ($lwork$).

$lwork \geq 0$ if $norm = 'M'$ or ' m ' (not referenced),
 $2 * nq0 + mp0 + ldw$ if $norm = '1'$, 'O' or 'o', 'I' or 'i',
where ldw is given by:

```
if( nprow.ne.npcol ) then
  ldw = mb_a * iceil(ceil(np0,mb_a), (lcm/nprow))
```

```

        else
          ldw = 0
        end if
        0 if norm = 'F', 'f', 'E' or 'e' (not referenced),
        where lcm is the least common multiple of nprow and npcol, lcm =
        ilcm( nprow, npcol ) and ceil(x,y) is a ScaLAPACK function that
        returns ceiling (x/y).
        iroffa = mod(ia-1, mb_a ), icoffa = mod( ja-1, nb_a ),
        iarow = indxg2p(ia, mb_a, myrow, rsrc_a, nprow),
        iacol = indxg2p(ja, nb_a, mycol, csrc_a, ncol),
        mp0 = numroc(m+iroffa, mb_a, myrow, iarow, nprow),
        nq0 = numroc(n+icoffa, nb_a, mycol, iacol, ncol),
        ilcm, ceil, indxg2p, and numroc are ScaLAPACK tool functions; myrow,
        mycol, nprow, and ncol can be determined by calling the subroutine
        blacs_gridinfo.

```

Output Parameters

<i>val</i>	The value returned by the routine.
------------	------------------------------------

p?lantr

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.

Syntax

Fortran:

```

val = pslantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pdlantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pclantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)
val = pzlantr(norm, uplo, diag, m, n, a, ia, ja, desca, work)

```

C:

```

float pslantr (char *norm , char *uplo , char *diag , MKL_INT *m , MKL_INT *n , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *work );
double pdlantr (char *norm , char *uplo , char *diag , MKL_INT *m , MKL_INT *n ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *work );
float pclantr (char *norm , char *uplo , char *diag , MKL_INT *m , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *work );
double pzlantr (char *norm , char *uplo , char *diag , MKL_INT *m , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *work );

```

Include Files

- C: mkl_scalapack.h

Description

The `p?lantr` routine returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$.

Input Parameters

<i>norm</i>	(global) CHARACTER. Specifies what value is returned by the routine: = 'M' or 'm': $\text{val} = \max(\text{abs}(A_{ij}))$, largest absolute value of the matrix A , it's not a matrix norm. = '1' or 'O' or 'o': $\text{val} = \text{norm1}(A)$, 1-norm of the matrix A (maximum column sum), = 'I' or 'i': $\text{val} = \text{normI}(A)$, infinity norm of the matrix A (maximum row sum), = 'F', 'f', 'E' or 'e': $\text{val} = \text{normF}(A)$, Frobenius norm of the matrix A (square root of sum of squares).
<i>uplo</i>	(global) CHARACTER. Specifies whether the upper or lower triangular part of the symmetric matrix $\text{sub}(A)$ is to be referenced. = 'U': Upper trapezoidal, = 'L': Lower trapezoidal. Note that $\text{sub}(A)$ is triangular instead of trapezoidal if $m = n$.
<i>diag</i>	(global) CHARACTER. Specifies whether the distributed matrix $\text{sub}(A)$ has unit diagonal. = 'N': Non-unit diagonal. = 'U': Unit diagonal.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. When $m = 0$, <code>p?lantr</code> is set to zero. $m \geq 0$.
<i>n</i>	(global) INTEGER. The number of columns to be operated on i.e the number of columns of the distributed submatrix $\text{sub}(A)$. When $n = 0$, <code>p?lantr</code> is set to zero. $n \geq 0$.
<i>a</i>	(local). REAL for <code>pslantr</code> DOUBLE PRECISION for <code>pdlantr</code> COMPLEX for <code>pclantr</code> COMPLEX*16 for <code>pzlantr</code> .

Pointer into the local memory to an array of size (lld_a , $LOCc(ja+n-1)$) containing the local pieces of the distributed matrix sub(A).

ia, ja

(global) INTEGER.

The row and column indices in the global array a indicating the first row and the first column of the submatrix sub(A), respectively.

desca

(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix A .

work

(local).

REAL for pslantr

DOUBLE PRECISION for pdlantr

COMPLEX for pclantr

COMPLEX*16 for pzlantr.

Array size ($iwork$).

$iwork \geq 0$ if $norm = 'M'$ or ' m ' (not referenced),

$nq0$ if $norm = '1'$, ' O ' or ' o ',

$mp0$ if $norm = 'I'$ or ' i ',

0 if $norm = 'F'$, ' f ', ' E ' or ' e ' (not referenced),

$iroffa = \text{mod}(ia-1, mb_a)$, $icoffa = \text{mod}(ja-1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$,

$iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol)$,

$mp0 = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow)$,

$nq0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcol)$,

`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

val

The value returned by the routine.

p?lapiv

Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.

Syntax

Fortran:

```
call pslapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip,
iwork)

call pdlapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip,
iwork)

call pclapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip,
iwork)
```

```
call pzlapiv(direc, rowcol, pivroc, m, n, a, ia, ja, desca, ipiv, ip, jp, descip,
iwork)
```

C:

```
void pslapiv (char *direc , char *rowcol , char *pivroc , MKL_INT *m , MKL_INT *n ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_INT *ip ,
MKL_INT *jp , MKL_INT *descip , MKL_INT *iwork );

void pdlapiv (char *direc , char *rowcol , char *pivroc , MKL_INT *m , MKL_INT *n ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_INT *ip ,
MKL_INT *jp , MKL_INT *descip , MKL_INT *iwork );

void pclapiv (char *direc , char *rowcol , char *pivroc , MKL_INT *m , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv , MKL_INT
*ip , MKL_INT *jp , MKL_INT *descip , MKL_INT *iwork );

void pzlapiv (char *direc , char *rowcol , char *pivroc , MKL_INT *m , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ipiv ,
MKL_INT *ip , MKL_INT *jp , MKL_INT *descip , MKL_INT *iwork );
```

Include Files

- C: mkl_scalapack.h

Description

The p?lapiv routine applies either P (permutation matrix indicated by *ipiv*) or $\text{inv}(P)$ to a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$, resulting in row or column pivoting. The pivot vector may be distributed across a process row or a column. The pivot vector should be aligned with the distributed matrix A . This routine will transpose the pivot vector, if necessary.

For example, if the row pivots should be applied to the columns of $\text{sub}(A)$, pass *rowcol*='C' and *pivroc*='C'.

Input Parameters

<i>direc</i>	(global) CHARACTER*1.
	Specifies in which order the permutation is applied:
	= 'F' (Forward). Applies pivots forward from top of matrix. Computes $P * \text{sub}(A)$.
	= 'B' (Backward). Applies pivots backward from bottom of matrix. Computes $\text{inv}(P) * \text{sub}(A)$.
<i>rowcol</i>	(global) CHARACTER*1.
	Specifies if the rows or columns are to be permuted:
	= 'R' Rows will be permuted,
	= 'C' Columns will be permuted.
<i>pivroc</i>	(global) CHARACTER*1.
	Specifies whether <i>ipiv</i> is distributed over a process row or column:
	= 'R' <i>ipiv</i> is distributed over a process row,
	= 'C' <i>ipiv</i> is distributed over a process column.
<i>m</i>	(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. When $m = 0$, p?lapiv is set to zero. $m \geq 0$.

n (global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. When $n = 0$, p?lapiv is set to zero. $n \geq 0$.

a (local).

REAL for pslapiv

DOUBLE PRECISION for pdlapiv

COMPLEX for pclapiv

COMPLEX*16 for pzlapiv .

Pointer into the local memory to an array of size $(\text{lld}_a, \text{LOCc}(\text{ja}+n-1))$ containing the local pieces of the distributed matrix $\text{sub}(A)$.

ia, ja (global) INTEGER.

The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca (global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix A .

ipiv (local) INTEGER.

Array, size ($lipiv$) ;

when $\text{rowcol}='R'$ or ' r ' :

$lipiv \geq \text{LOCr}(\text{ia}+m-1) + mb_a$ if $\text{pivroc}='C'$ or ' c ',

$lipiv \geq \text{LOCc}(m + \text{mod}(jp-1, nb_p))$ if $\text{pivroc}='R'$ or ' r ', and,

when $\text{rowcol}='C'$ or ' c ' :

$lipiv \geq \text{LOCr}(n + \text{mod}(ip-1, mb_p))$ if $\text{pivroc}='C'$ or ' c ',

$lipiv \geq \text{LOCc}(\text{ja}+n-1) + nb_a$ if $\text{pivroc}='R'$ or ' r '.

This array contains the pivoting information. $ipiv(i)$ is the global row (column), local row (column) i was swapped with. When $\text{rowcol}='R'$ or ' r ' and $\text{pivroc}='C'$ or ' c ', or $\text{rowcol}='C'$ or ' c ' and $\text{pivroc}='R'$ or ' r ', the last piece of this array of size mb_a (resp. nb_a) is used as workspace. In those cases, this array is tied to the distributed matrix A .

ip, jp (global) INTEGER. The row and column indices in the global array P indicating the first row and the first column of the submatrix $\text{sub}(P)$, respectively.

descip (global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed vector $ipiv$.

iwork (local). INTEGER.

Array, size (ldw), where ldw is equal to the workspace necessary for transposition, and the storage of the transposed $ipiv$:

Let lcm be the least common multiple of $nprow$ and $npcol$.

```

If( rowcol.eq.'r' .and. pivroc. eq.'r') then
    If( nprow.eq. npcol) then
        ldw = LOCr( n_p + mod(jp-1, nb_p) ) + nb_p
    else
        ldw = LOCr( n_p + mod(jp-1, nb_p) ) +
            nb_p * ceil( ceil(LOCc(n_p)/nb_p) / (lcm/npcol) )
    end if
else if( rowcol.eq.'c' .and. pivroc.eq.'c') then
    if( nprow.eq.
        npcol ) then
        ldw = LOCc( m_p + mod(ip-1, mb_p) ) + mb_p
    else
        ldw = LOCc( m_p + mod(ip-1, mb_p) ) +
            mb_p *ceil(ceil(LOCr(m_p)/mb_p) / (lcm/nprow) )
    end if
else
    iwork is not referenced.
end if.
```

Output Parameters

a (local).

On exit, the local pieces of the permuted distributed submatrix.

p?laqge

Scales a general rectangular matrix, using row and column scaling factors computed by p?geequ .

Syntax

Fortran:

```
call pslaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pdlaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pclaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
call pzlaqge(m, n, a, ia, ja, desca, r, c, rowcnd, colcnd, amax, equed)
```

C:

```
void pslaqge (MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *r , float *c , float *rowcnd , float *colcnd , float *amax , char
*equed );
```

```

void pdlaqge (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *r , double *c , double *rowcnd , double *colcnd , double
*amax , char *equed );

void pclaqge (MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *r , float *c , float *rowcnd , float *colcnd , float *amax ,
char *equed );

void pzlaqge (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *r , double *c , double *rowcnd , double *colcnd , double
*amax , char *equed );

```

Include Files

- C: mkl_scalapack.h

Description

The p?laqge routine equilibrates a general m -by- n distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ using the row and scaling factors in the vectors r and c computed by p?geequ.

Input Parameters

<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>a</i>	(local). REAL for psraqge DOUBLE PRECISION for pdlaqge COMPLEX for pclaqge COMPLEX*16 for pzlaqge. Pointer into the local memory to an array of size (lld_a, LOCc(ja+n-1)). On entry, this array contains the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array, size (dlen_). The array descriptor for the distributed matrix A .
<i>r</i>	(local). REAL for psraqge DOUBLE PRECISION for pdlaqge COMPLEX for pclaqge COMPLEX*16 for pzlaqge.

Array, size $\text{LOC}_r(m_a)$. The row scale factors for $\text{sub}(A)$. r is aligned with the distributed matrix A , and replicated across every process column. r is tied to the distributed matrix A .

c (local).

REAL **for** `pslaqge`
 DOUBLE PRECISION **for** `pdlaqge`
 COMPLEX **for** `pclaqge`
 COMPLEX*16 **for** `pzlaqge`.

Array, size $\text{LOC}_c(n_a)$. The row scale factors for $\text{sub}(A)$. c is aligned with the distributed matrix A , and replicated across every process column. c is tied to the distributed matrix A .

rowcnd (local).

REAL **for** `pslaqge`
 DOUBLE PRECISION **for** `pdlaqge`
 COMPLEX **for** `pclaqge`
 COMPLEX*16 **for** `pzlaqge`.

The global ratio of the smallest $r(i)$ to the largest $r(i)$, $ia \leq i \leq ia+m-1$.

colcnd (local).

REAL **for** `pslaqge`
 DOUBLE PRECISION **for** `pdlaqge`
 COMPLEX **for** `pclaqge`
 COMPLEX*16 **for** `pzlaqge`.

The global ratio of the smallest $c(i)$ to the largest $c(i)$, $ia \leq i \leq ia+n-1$.

amax (global). REAL **for** `pslaqge`

DOUBLE PRECISION **for** `pdlaqge`
 COMPLEX **for** `pclaqge`
 COMPLEX*16 **for** `pzlaqge`.

Absolute value of largest distributed submatrix entry.

Output Parameters

a (local).

On exit, the equilibrated distributed matrix. See `equed` for the form of the equilibrated distributed submatrix.

equed (global) CHARACTER.

Specifies the form of equilibration that was done.

= 'N': No equilibration

= 'R': Row equilibration, that is, $\text{sub}(A)$ has been pre-multiplied by `diag(r(ia:ia+m-1))`,

```
= 'C': column equilibration, that is, sub(A) has been post-multiplied by
diag(c(ja:ja+n-1)),
= 'B': Both row and column equilibration, that is, sub(A) has been
replaced by diag(r(ia:ia+m-1))* sub(A) * diag(c(ja:ja+n-1)).
```

p?laqr0

Computes the eigenvalues of a Hessenberg matrix and optionally returns the matrices from the Schur decomposition.

Syntax

Fortran:

```
call pslaqr0( wantt, wantz, n, ilo, ihi, h, desch, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, liwork, info, reclevel )
call pdlaqr0( wantt, wantz, n, ilo, ihi, h, desch, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, liwork, info, reclevel )
```

Include Files

- C: mkl_scalapack.h

Description

p?laqr0 computes the eigenvalues of a Hessenberg matrix H and, optionally, the matrices T and Z from the Schur decomposition $H = Z*T*Z^T$, where T is an upper quasi-triangular matrix (the Schur form), and Z is the orthogonal matrix of Schur vectors.

Optionally Z may be postmultiplied into an input orthogonal matrix Q so that this routine can give the Schur factorization of a matrix A which has been reduced to the Hessenberg form H by the orthogonal matrix Q : $A = Q * H * Q^T = (QZ) * T * (QZ)^T$.

Input Parameters

<i>wantt</i>	(global) LOGICAL = .TRUE. : the full Schur form T is required; = .FALSE. : only eigenvalues are required.
<i>wantz</i>	(global) LOGICAL = .TRUE. : the matrix of Schur vectors Z is required; = .FALSE.: Schur vectors are not required.
<i>n</i>	(global) INTEGER The order of the Hessenberg matrix H (and Z if <i>wantz</i>). $n \geq 0$.
<i>ilo, ihi</i>	(global) INTEGER It is assumed that <i>h</i> is already upper triangular in rows and columns 1: <i>ilo-1</i> and <i>ihi+1:n</i> . <i>ilo</i> and <i>ihi</i> are normally set by a previous call to p?gebal, and then passed to p?gehrd when the matrix output by <i>ihi</i> is reduced to Hessenberg form. Otherwise <i>ilo</i> and <i>ihi</i> should be set to 1 and <i>n</i> , respectively. If <i>n > 0</i> , then $1 \leq ilo \leq ihi \leq n$.

If $n = 0$, then $\text{ilo} = 1$ and $\text{ih} = 0$.

h	REAL for pslaqr0 DOUBLE PRECISION for pdlaqr0 (global) array, dimension ($\text{ld}_h, \text{LOC}_c(n)$) The upper Hessenberg matrix H .
desc_h	(global and local) INTEGER array of dimension $dlen_$. The array descriptor for the distributed matrix H .
$\text{ilo}_z, \text{ih}_z$	INTEGER Specify the rows of z to which transformations must be applied if want_z is .TRUE., $1 \leq \text{ilo}_z \leq \text{ilo}; \text{ih}_z \leq \text{ih}_z \leq n$.
z	REAL for pslaqr0 DOUBLE PRECISION for pdlaqr0 Array, dimension ($\text{ld}_z, \text{LOC}_c(n)$). If want_z is .TRUE., contains the matrix Z . If want_z is .FALSE., z is not referenced.
desc_z	(global and local) INTEGER array of dimension $dlen_$. The array descriptor for the distributed matrix Z .
work	REAL for pslaqr0 DOUBLE PRECISION for pdlaqr0 (local workspace) array, dimension(lwork)
lwork	(local) INTEGER The length of the workspace array work .
iwork	(local workspace) INTEGER array, dimension (liwork)
liwork	(local) INTEGER The length of the workspace array iwork .
reclevel	(local) INTEGER Level of recursion. $\text{reclevel} = 0$ must hold on entry.

OUTPUT Parameters

h	On exit, if want_t is .TRUE., h is upper quasi-triangular in rows and columns $\text{ilo}:ih$, with 1-by-1 and 2-by-2 blocks on the main diagonal. The 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with $h(i,i) = h(i+1,i+1)$ and $h(i+1,i)*h(i,i+1) < 0$. If $\text{info} = 0$ and want_t is .FALSE., the contents of h are unspecified on exit.
wr, wi	REAL for pslaqr0

DOUBLE PRECISION for pdlaqr0

The real and imaginary parts, respectively, of the computed eigenvalues ilo to ahi are stored in the corresponding elements of wr and wi . If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of wr and wi , say the i -th and $(i+1)$ -th, with $wi(i) > 0$ and $wi(i+1) < 0$. If $wantt$ is .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in h .

z Updated matrix with transformations applied only to the submatrix $z(ilo:ahi, ilo:ahi)$.

If $COMPZ = 'I'$, on exit, if $info = 0$, z contains the orthogonal matrix Z of the Schur vectors of H .

If $wantz$ is .TRUE., then $z(ilo:ahi, iloz:ihiz)$ is replaced by $z(ilo:ahi, iloz:ihiz)*U$, when U is the orthogonal/unitary Schur factor of $h(ilo:ahi, ilo:ahi)$.

If $wantz$ is .FALSE., then z is not defined.

$work(1)$ On exit, if $info = 0$, $work(1)$ returns the optimal $lwork$.

$iwork(1)$ On exit, if $info = 0$, $iwork(1)$ returns the optimal $liwork$.

$info$ INTEGER

> 0: if $info = i$, then the routine failed to compute all the eigenvalues. Elements $1:ilo-1$ and $i+1:n$ of wr and wi contain those eigenvalues which have been successfully computed.

> 0: if $wantt$ is .FALSE., then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns ilo through ilo of the final output value of h .

> 0: if $wantt$ is .TRUE., then $(\text{initial value of } h)*U = U*(\text{final value of } h)$, where U is an orthogonal/unitary matrix. The final value of h is upper Hessenberg and quasi-triangular/triangular in rows and columns $info+1$ through ahi .

> 0: if $wantz$ is .TRUE., then $(\text{final value of } z(ilo:ahi, iloz:ihiz)) = (\text{initial value of } z(ilo:ahi, iloz:ihiz))*U$, where U is the orthogonal/unitary matrix in the previous expression (regardless of the value of $wantt$).

> 0: if $wantz$ is .FALSE., then z is not accessed.

p?laqr1

Sets a scalar multiple of the first column of the product of a 2-by-2 or 3-by-3 matrix and specified shifts.

Syntax

Fortran:

```
call pslaqr1( wantt, wantz, n, ilo, ahi, a, desca, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, ilwork, info )
```

```
call pdlaqr1( wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, work,
lwork, iwork, ilwork, info )
```

Include Files

- C: mkl_scalapack.h

Description

`p?laqr1` is an auxiliary routine used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns `ilo` to `ihi`.

This is a modified version of `p?lahqr` from ScaLAPACK version 1.7.3. The following modifications were made:

- Workspace query functionality was added.
- Aggressive early deflation is implemented.
- Aggressive deflation (looking for two consecutive small subdiagonal elements by PSLACONSB) is abandoned.
- The returned Schur form is now in canonical form, i.e., the returned 2-by-2 blocks really correspond to complex conjugate pairs of eigenvalues.
- For some reason, the original version of `p?lahqr` sometimes did not read out the converged eigenvalues correctly. This is now fixed.

Input Parameters

<code>wantt</code>	(global) LOGICAL
	= .TRUE. : the full Schur form T is required;
	= .FALSE.: only eigenvalues are required.
<code>wantz</code>	(global) LOGICAL
	= .TRUE. : the matrix of Schur vectors Z is required;
	= .FALSE.: Schur vectors are not required.
<code>n</code>	(global) LOGICAL
	The order of the Hessenberg matrix A (and Z if <code>wantz</code>). $n \geq 0$.
<code>ilo, ihi</code>	(global) INTEGER
	It is assumed that <code>a</code> is already upper quasi-triangular in rows and columns $ihi+1:n$, and that $a(ilo,ilo-1) = 0$ (unless <code>ilo = 1</code>). <code>p?laqr1</code> works primarily with the Hessenberg submatrix in rows and columns <code>ilo</code> to <code>ihi</code> , but applies transformations to all of H if <code>wantt</code> is .TRUE..
	$1 \leq ilo \leq \max(1,ihi); ihi \leq n$.
<code>a</code>	REAL for pslaqr1 DOUBLE PRECISION for pdlaqr1
	(global) array, dimension (<code>desca(Ild_),LOC_c(n)</code>)
	On entry, the upper Hessenberg matrix A .
<code>desca</code>	(global and local) INTEGER array of dimension <code>dlen_</code> .
	The array descriptor for the distributed matrix A .

<i>iloz, ihiz</i>	(global) INTEGER Specify the rows of <i>z</i> to which transformations must be applied if <i>wantz</i> is .TRUE.. $1 \leq iloz \leq ilo; ihi \leq ihiz \leq n.$
<i>z</i>	REAL for pslaqr1 DOUBLE PRECISION for pdlaqr1 (global) array, dimension (<i>descz</i> (<i>lld_</i>), <i>LOC_c(n)</i>). If <i>wantz</i> is .TRUE., on entry <i>z</i> must contain the current matrix <i>Z</i> of transformations accumulated by <i>p?hseqr</i> If <i>wantz</i> is .FALSE., <i>z</i> is not referenced.
<i>descz</i>	(global and local) INTEGER array of dimension <i>dlen_</i> . The array descriptor for the distributed matrix <i>Z</i> .
<i>work</i>	REAL for pslaqr1 DOUBLE PRECISION for pdlaqr1 (local output) array of size <i>lwork</i>
<i>lwork</i>	(local) INTEGER The size of the work array (<i>lwork</i> >=1). If <i>lwork</i> =-1, then a workspace query is assumed.
<i>iwork</i>	(global and local) INTEGER array of size <i>ilwork</i> This holds the some of the IBLK integer arrays.
<i>ilwork</i>	(local) INTEGER The size of the <i>iwork</i> array (<i>ilwork</i> ≥3).

OUTPUT Parameters

<i>a</i>	If <i>wantt</i> is .TRUE., <i>a</i> is upper quasi-triangular in rows and columns <i>ilo:ihiz</i> , with any 2-by-2 or larger diagonal blocks not yet in standard form. If <i>wantt</i> is .FALSE., the contents of <i>a</i> are unspecified on exit.
<i>wr, wi</i>	REAL for pslaqr1 DOUBLE PRECISION for pdlaqr1 (global replicated) array, dimension (<i>n</i>) The real and imaginary parts, respectively, of the computed eigenvalues <i>ilo</i> to <i>ihiz</i> are stored in the corresponding elements of <i>wr</i> and <i>wi</i> . If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of <i>wr</i> and <i>wi</i> , say the <i>i</i> -th and (<i>i</i> +1)th, with <i>wi(i)</i> > 0 and <i>wi(i+1)</i> < 0. If <i>wantt</i> is .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in <i>a</i> . <i>a</i> may be returned with larger diagonal blocks until the next release.

z On exit *z* is updated; transformations are applied only to the submatrix *z(iloz:ihiz,ilo:ihi)*.
If *wantz* is .FALSE., *z* is not referenced.

work(1) On exit, if *info* = 0, *work(1)* returns the optimal *lwork*.

info (global) INTEGER
< 0: parameter number -*info* incorrect or inconsistent
= 0: successful exit
> 0: p?laqr1 failed to compute all the eigenvalues *ilo* to *ihiz* in a total of $30*(ihiz-ilo+1)$ iterations; if *info* = *i*, elements *i+1:ihiz* of *wr* and *wi* contain those eigenvalues which have been successfully computed.

Application Notes

This algorithm is very similar to p?lahqr. Unlike p?lahqr, instead of sending one double shift through the largest unreduced submatrix, this algorithm sends multiple double shifts and spaces them apart so that there can be parallelism across several processor row/columns. Another critical difference is that this algorithm aggregates multiple transforms together in order to apply them in a block fashion.

Current Notes and/or Restrictions:

- This code requires the distributed block size to be square and at least six (6); unlike simpler codes like LU, this algorithm is extremely sensitive to block size. Unwise choices of too small a block size can lead to bad performance.
- This code requires *a* and *z* to be distributed identically and have identical contexts.
- This release currently does not have a routine for resolving the Schur blocks into regular 2x2 form after this code is completed. Because of this, a significant performance impact is required while the deflation is done by sometimes a single column of processors.
- This code does not currently block the initial transforms so that none of the rows or columns for any bulge are completed until all are started. To offset pipeline start-up it is recommended that at least $2*LCM(NPROW,NPCOL)$ bulges are used (if possible)
- The maximum number of bulges currently supported is fixed at 32. In future versions this will be limited only by the incoming *work* array.
- The matrix *A* must be in upper Hessenberg form. If elements below the subdiagonal are nonzero, the resulting transforms may be nonsimilar. This is also true with the LAPACK routine.
- For this release, it is assumed *rsrc_*=*csrc_*=0
- Currently, all the eigenvalues are distributed to all the nodes. Future releases will probably distribute the eigenvalues by the column partitioning.
- The internals of this routine are subject to change.

p?laqr2

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

Fortran:

```
call pslaqr2( wantt, wantz, n, ktop, kbot, nw, a, desca, iloz, ihiz, z, descz, ns, nd,
sr, si, t, ldt, v, ldv, wr, wi, work, lwork )
```

```
call pdlaqr2( wantt, wantz, n, ktop, kbot, nw, a, desca, iloz, ihiz, z, descz, ns, nd,
sr, si, t, ldt, v, ldv, wr, wi, work, lwork )
```

Include Files

- C: mkl_scalapack.h

Description

p?laqr2 accepts as input an upper Hessenberg matrix A and performs an orthogonal similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output A is overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal similarity transformation of A . It is to be hoped that the final version of A has many zero subdiagonal entries.

This routine handles small deflation windows which is affordable by one processor. Normally, it is called by p?laqr1. All the inputs are assumed to be valid without checking.

Input Parameters

wantt	(global) LOGICAL
	If .TRUE., then the Hessenberg matrix A is fully updated so that the quasi-triangular Schur factor may be computed (in cooperation with the calling subroutine).
	If .FALSE., then only enough of A is updated to preserve the eigenvalues.
wantz	(global) LOGICAL
	If .TRUE., then the orthogonal matrix Z is updated so that the orthogonal Schur factor may be computed (in cooperation with the calling subroutine).
	If .FALSE., then z is not referenced.
n	(global) INTEGER
	The order of the matrix A and (if wantz is .TRUE.) the order of the orthogonal matrix Z .
ktop, kbot	(global) INTEGER
	It is assumed without a check that either $kbot = n$ or $A(kbot+1,kbot)=0$. $kbot$ and $ktop$ together determine an isolated block along the diagonal of the Hessenberg matrix. However, $A(ktop,ktop-1)=0$ is not essentially necessary if wantt is .TRUE. .
nw	(global) INTEGER
	Deflation window size. $1 \leq nw \leq (kbot-ktop+1)$. Normally $nw \geq 3$ if p?laqr2 is called by p?laqr1.
a	REAL for pslaqr2 DOUBLE PRECISION for pdlaqr2 (local) array, dimension (desca(/ld_),LOC_c(n))
	The initial n -by- n section of a stores the Hessenberg matrix undergoing aggressive early deflation.
desca	(global and local) INTEGER array of dimension DLEN_.

The array descriptor for the distributed matrix A .

iloz, ihiz

(global) INTEGER

Specify the rows of z to which transformations must be applied if *wantz* is .TRUE.. $1 \leq iloz \leq ihiz \leq n$.

z

REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

Array, dimension (*descz*(*ldz*),*LOCc*(*n*))

If *wantz* is .TRUE., then on output, the orthogonal similarity transformation mentioned above has been accumulated into *z*(*iloz*:*ihiz*, *kbot*:*ktop*) from the right.

If *wantz* is .FALSE., then *z* is unreferenced.

descz

(global and local) INTEGER array of dimension DLEN_.

The array descriptor for the distributed matrix Z .

t

REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

(local workspace) array, dimension *ldt***nw*.

ldt

(local) INTEGER

The leading dimension of the array *t*. *ldt* $\geq nw$.

v

REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

(local workspace) array, dimension *ldv***nw*.

ldv

(local) INTEGER

The leading dimension of the array *v*. *ldv* $\geq nw$.

wr, wi

REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

(local workspace) array, dimension *kbot*.

work

REAL for pslaqr2

DOUBLE PRECISION for pdlaqr2

(local workspace) array, dimension *lwork*.

lwork

(local) INTEGER

work(lwork) is a local array and *lwork* is assumed big enough so that *lwork* $\geq nw * nw$.

OUTPUT Parameters

<i>a</i>	On output <i>a</i> has been transformed by an orthogonal similarity transformation, perturbed, and returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.
<i>z</i>	
<i>ns</i>	(global) INTEGER The number of unconverged (ie approximate) eigenvalues returned in <i>sr</i> and <i>si</i> that may be used as shifts by the calling subroutine.
<i>nd</i>	(global) INTEGER The number of converged eigenvalues uncovered by this subroutine.
<i>sr, si</i>	REAL for pslaqr2 DOUBLE PRECISION for pdlaqr2 (global) array, dimension <i>kbot</i> On output, the real and imaginary parts of approximate eigenvalues that may be used for shifts are stored in <i>sr(kbot-nd-ns+1)</i> through <i>sr(kbot-nd)</i> and <i>si(kbot-nd-ns+1)</i> through <i>si(kbot-nd)</i> , respectively. On processor #0, the real and imaginary parts of converged eigenvalues are stored in <i>sr(kbot-nd+1)</i> through <i>sr(kbot)</i> and <i>si(kbot-nd+1)</i> through <i>si(kbot)</i> , respectively. On other processors, these entries are set to zero.

p?laqr3

Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax

Fortran:

```
call pslaqr3( wantt, wantz, n, ktop, kbot, nw, h, desch, iloz, ihiz, z, descz, ns, nd,
sr, si, v, descv, nh, t, desct, nv, wv, descw, work, lwork, iwork, liwork, reclevel )
call pdlaqr3( wantt, wantz, n, ktop, kbot, nw, h, desch, iloz, ihiz, z, descz, ns, nd,
sr, si, v, descv, nh, t, desct, nv, wv, descw, work, lwork, iwork, liwork, reclevel )
```

Include Files

- C: mkl_scalapack.h

Description

This subroutine accepts as input an upper Hessenberg matrix *H* and performs an orthogonal similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output *H* is overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal similarity transformation of *H*. It is to be hoped that the final version of *H* has many zero subdiagonal entries.

Input Parameters

wantt (global) LOGICAL
 If .TRUE., then the Hessenberg matrix *h* is fully updated so that the quasi-triangular Schur factor may be computed (in cooperation with the calling subroutine).
 If .FALSE., then only enough of *h* is updated to preserve the eigenvalues.

wantz (global) LOGICAL
 If .TRUE., then the orthogonal matrix *z* is updated so that the orthogonal Schur factor may be computed (in cooperation with the calling subroutine).
 If .FALSE., then *z* is not referenced.

n (global) INTEGER
 The order of the matrix *H* and (if *wantz* is .TRUE.) the order of the orthogonal matrix *Z*.

ktop (global) INTEGER
 It is assumed that either *ktop* = 1 or *h(ktop,ktop-1)*=0. *kbot* and *ktop* together determine an isolated block along the diagonal of the Hessenberg matrix.

kbot (global) INTEGER
 It is assumed without a check that either *kbot* = *n* or *h(kbot+1,kbot)*=0. *kbot* and *ktop* together determine an isolated block along the diagonal of the Hessenberg matrix.

nw (global) INTEGER
 Deflation window size. $1 \leq nw \leq (kbot-ktop+1)$.

h REAL for pslaqr3
 DOUBLE PRECISION for pdlaqr3
 (local) array, dimension (*desc*h**(*lld_*),*LOCc(n)*)
 The initial *n*-by-*n* section of *h* stores the Hessenberg matrix undergoing aggressive early deflation.

*desc*h** (global and local) INTEGER array of dimension *dlen_*.
 The array descriptor for the distributed matrix *h*.

iloz, ihiz (global) INTEGER
 Specify the rows of *z* to which transformations must be applied if *wantz* is .TRUE.. $1 \leq iloz \leq ihiz \leq n$.

z REAL for pslaqr3
 DOUBLE PRECISION for pdlaqr3
 Array, dimension (*desc*z**(*lld_*),*LOCc(n)*)

If `wantz` is .TRUE., then on output, the orthogonal similarity transformation mentioned above has been accumulated into $z(iloz:ihiz,kbot:ktop)$ from the right.

If `wantz` is .FALSE., then z is unreferenced.

`descz` (global and local) INTEGER array of dimension $dlen_$.

The array descriptor for the distributed matrix z .

`v` REAL for `pslaqr3`

DOUBLE PRECISION for `pdlaqr3`

(global workspace) array, dimension ($descv(lld_), LOC_c(nw)$)

An nw -by- nw distributed work array.

`descv` (global and local) INTEGER array of dimension $dlen_$.

The array descriptor for the distributed matrix v .

`nh` INTEGER scalar

The number of columns of t . $nh \geq nw$.

`t` REAL for `pslaqr3`

DOUBLE PRECISION for `pdlaqr3`

(global workspace) array, dimension ($desc(t)(lld_), LOC_c(nh)$)

`desc(t)` (global and local) INTEGER array of dimension $dlen_$.

The array descriptor for the distributed matrix t .

`nv` (global) INTEGER

The number of rows of work array wv available for workspace. $nv \geq nw$.

`wv` (global workspace) REAL array, dimension

($descw(lld_), LOC_c(nw)$)

`descw` (global and local) INTEGER array of dimension $dlen_$.

The array descriptor for the distributed matrix wv .

`work` (local workspace) REAL array, dimension $lwork$.

`lwork` (local) INTEGER

The dimension of the work array `work` ($lwork \geq 1$). $lwork = 2 * nw$ suffices, but greater efficiency may result from larger values of `lwork`.

If `lwork` = -1, then a workspace query is assumed; `p?laqr3` only estimates the optimal workspace size for the given values of n , nw , $ktop$ and $kbot$. The estimate is returned in `work(1)`. No error message related to `lwork` is issued by `xerbla`. Neither `h` nor `z` are accessed.

`iwork` (local workspace) INTEGER array, dimension ($liwork$)

`liwork` (local) INTEGER

The length of the workspace array $iwork$ ($liwork \geq 1$).

If $liwork=-1$, then a workspace query is assumed.

OUTPUT Parameters

h	On output h has been transformed by an orthogonal similarity transformation, perturbed, and returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.
z	IF $wantz$ is .TRUE., then on output, the orthogonal similarity transformation mentioned above has been accumulated into $z(iloz:ihiz,kbot:ktop)$ from the right. If $wantz$ is .FALSE., then z is unreferenced.
ns	(global) INTEGER The number of unconverged (ie approximate) eigenvalues returned in sr and si that may be used as shifts by the calling subroutine.
nd	(global) INTEGER The number of converged eigenvalues uncovered by this subroutine.
sr, si	REAL for pslaqr3 DOUBLE PRECISION for pdlaqr3 (global) array, dimension $kbot$ The real and imaginary parts of approximate eigenvalues that may be used for shifts are stored in $sr(kbot-nd-ns+1)$ through $sr(kbot-nd)$ and $si(kbot-nd-ns+1)$ through $si(kbot-nd)$, respectively. The real and imaginary parts of converged eigenvalues are stored in $sr(kbot-nd+1)$ through $sr(kbot)$ and $si(kbot-nd+1)$ through $si(kbot)$, respectively.
$work(1)$	On exit, if $info = 0$, $work(1)$ returns the optimal $lwork$
$iwork(1)$	On exit, if $info = 0$, $iwork(1)$ returns the optimal $liwork$

p?laqr4

Computes the eigenvalues of a Hessenberg matrix, and optionally computes the matrices from the Schur decomposition.

Syntax

Fortran:

```
call pslaqr4( wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, t,
              ldt, v, ldv, work, lwork, info )
call pdlaqr4( wantt, wantz, n, ilo, ihi, a, desca, wr, wi, iloz, ihiz, z, descz, t,
              ldt, v, ldv, work, lwork, info )
```

Include Files

- C: mkl_scalapack.h

Description

`p?laqr4` is an auxiliary routine used to find the Schur decomposition and or eigenvalues of a matrix already in Hessenberg form from cols ilo to ih_i . This routine requires that the active block is small enough, i.e. $ih_i - ilo + 1 \leq ldt$, so that it can be solved by LAPACK. Normally, it is called by `p?laqr1`. All the inputs are assumed to be valid without checking.

Input Parameters

<code>wantt</code>	(global) LOGICAL = .TRUE. : the full Schur form T is required; = .FALSE.: only eigenvalues are required.
<code>wantz</code>	(global) LOGICAL = .TRUE. : the matrix of Schur vectors Z is required; = .FALSE.: Schur vectors are not required.
<code>n</code>	(global) INTEGER The order of the Hessenberg matrix A (and Z if <code>wantz</code>). $n \geq 0$.
<code>ilo, ih_i</code>	(global) INTEGER It is assumed that a is already upper quasi-triangular in rows and columns $ih_i + 1:n$, and that $a(ilo,ilo-1) = 0$ (unless $ilo = 1$). <code>p?laqr4</code> works primarily with the Hessenberg submatrix in rows and columns ilo to ih_i , but applies transformations to all of A if <code>wantt</code> is .TRUE.. $1 \leq ilo \leq \max(1,ih_i); ih_i \leq n$.
<code>a</code>	REAL for <code>pslaqr4</code> DOUBLE PRECISION for <code>pdlaqr4</code> (global) array, dimension (<code>desca(Ild_),LOC_c(n)</code>) The upper Hessenberg matrix A .
<code>desca</code>	(global and local) INTEGER array of dimension <code>dlen_</code> . The array descriptor for the distributed matrix a .
<code>iloz, ihiz</code>	(global) INTEGER Specify the rows of z to which transformations must be applied if <code>wantz</code> is .TRUE.. $1 \leq iloz \leq ilo; ih_i \leq ihiz \leq n$.
<code>z</code>	REAL for <code>pslaqr4</code> DOUBLE PRECISION for <code>pdlaqr4</code> (global) array. If <code>wantz</code> is .TRUE., on entry z must contain the current matrix Z of transformations accumulated by <code>p?hseqr</code> . If <code>wantz</code> is .FALSE., z is not referenced.
<code>descz</code>	(global and local) INTEGER array of dimension <code>dlen_</code> .

The array descriptor for the distributed matrix z .

t REAL for pslaqr4
 DOUBLE PRECISION for pdlaqr4
 (local workspace) array, dimension $l_{dt}*(i_{hi}-i_{lo}+1)$.

l_{dt} (local) INTEGER
 The leading dimension of the array t . $l_{dt} \geq i_{hi}-i_{lo}+1$.

v REAL for pslaqr4
 DOUBLE PRECISION for pdlaqr4
 (local workspace) array, dimension $l_{dv}*(i_{hi}-i_{lo}+1)$.

l_{dv} (local) INTEGER
 The leading dimension of the array v . $l_{dv} \geq i_{hi}-i_{lo}+1$.

$work$ REAL for pslaqr4
 DOUBLE PRECISION for pdlaqr4
 (local workspace) array, dimension l_{work} .

l_{work} (local) INTEGER
 The dimension of the work array $work$.
 $l_{work} \geq i_{hi}-i_{lo}+1$.

OUTPUT Parameters

a On exit, if $wantt$ is .TRUE., a is upper quasi-triangular in rows and columns $i_{lo}:i_{hi}$, with any 2-by-2 or larger diagonal blocks not yet in standard form. If $wantt$ is .FALSE., the contents of a are unspecified on exit.

wr, wi REAL for pslaqr4
 DOUBLE PRECISION for pdlaqr4
 (global replicated) array, dimension (n)
 The real and imaginary parts, respectively, of the computed eigenvalues i_{lo} to i_{hi} are stored in the corresponding elements of wr and wi . If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of wr and wi , say the i -th and $(i+1)$ -th, with $wi(i) > 0$ and $wi(i+1) < 0$. If $wantt$ is .TRUE., the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in a . a may be returned with larger diagonal blocks until the next release.

z If $wantz$ is .TRUE., z is updated with transformations applied only to the submatrix $z(i_{lo}:i_{hi}, i_{lo}:i_{hi})$.

$info$ (global) INTEGER
 < 0: parameter number $-info$ incorrect or inconsistent;

= 0: successful exit;
> 0: p?laqr4 failed to compute all the eigenvalues ilo to ih_i in a total of $30*(ih_i-ilo+1)$ iterations; if $info = i$, elements $i+1:ih_i$ of wr and wi contain those eigenvalues which have been successfully computed.

p?laqr5

Performs a single small-bulge multi-shift QR sweep.

Syntax

Fortran:

```
call pslaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, desch, iloz,
ihiz, z, descz, work, lwork, iwork, liwork )

call pdlaqr5( wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, desch, iloz,
ihiz, z, descz, work, lwork, iwork, liwork )
```

Include Files

- C: mkl_scalapack.h

Description

This auxiliary subroutine called by p?laqr0 performs a single small-bulge multi-shift QR sweep by chasing separated groups of bulges along the main block diagonal of h .

Input Parameters

<i>wantt</i>	(global) LOGICAL scalar
	<i>wantt</i> = .TRUE. if the quasi-triangular Schur factor is being computed. <i>wantt</i> is set to .FALSE. otherwise.
<i>wantz</i>	(global) LOGICAL scalar
	<i>wantz</i> = .TRUE. if the orthogonal Schur factor is being computed. <i>wantz</i> is set to .FALSE. otherwise.
<i>kacc22</i>	(global) INTEGER
	Value 0, 1, or 2. Specifies the computation mode of far-from-diagonal orthogonal updates. = 0: p?laqr5 does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries. = 1: p?laqr5 accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries. = 2: p?laqr5 accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.
<i>n</i>	(global) INTEGER scalar
	The order of the Hessenberg matrix h and, if <i>wantz</i> is .TRUE., the order of the orthogonal matrix Z .

<i>ktop, kbot</i>	(global) INTEGER scalar
	These are the first and last rows and columns of an isolated diagonal block upon which the QR sweep is to be applied. It is assumed without a check that either $ktop = 1$ or $h(ktop,ktop-1) = 0$ and either $kbot = n$ or $h(kbot+1,kbot) = 0$.
<i>nshfts</i>	(global) INTEGER scalar
	<i>nshfts</i> gives the number of simultaneous shifts. <i>nshfts</i> must be positive and even.
<i>sr, si</i>	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (global) Array of size (<i>nshfts</i>) <i>sr</i> contains the real parts and <i>si</i> contains the imaginary parts of the <i>nshfts</i> shifts of origin that define the multi-shift QR sweep.
<i>h</i>	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (local) Array of size (<i>desch(lld_)</i> , <i>LOC_c(n)</i>) On input <i>h</i> contains a Hessenberg matrix.
<i>descch</i>	(global and local) INTEGER array of dimension <i>dlen_</i> . The array descriptor for the distributed matrix <i>h</i> .
<i>iloz, ihiz</i>	(global) INTEGER Specify the rows of <i>z</i> to which transformations must be applied if <i>wantz</i> is .TRUE.. $1 \leq iloz \leq ihiz \leq n$
<i>z</i>	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (local) array of size (<i>descz(lld_)</i> , <i>LOC_c(n)</i>) If <i>wantz</i> = .TRUE., then the QR Sweep orthogonal similarity transformation is accumulated into <i>z(iloz:ihiz,kbot:ktop)</i> from the right. If <i>wantz</i> = .FALSE., then <i>z</i> is unreferenced.
<i>descz</i>	(global and local) INTEGER array of dimension <i>dlen_</i> . The array descriptor for the distributed matrix <i>z</i> .
<i>work</i>	REAL for pslaqr5 DOUBLE PRECISION for pdlaqr5 (local workspace) array, dimension(<i>lwork</i>)
<i>lwork</i>	(local) INTEGER The size of the <i>work</i> array (<i>lwork</i> ≥ 1).

If $lwork=-1$, then a workspace query is assumed.

iwork (local workspace) INTEGER array, dimension (*lwork*)

lwork (local) INTEGER

The size of the *iwork* array ($lwork \geq 1$).

If $lwork=-1$, then a workspace query is assumed.

Output Parameters

h A multi-shift QR sweep with shifts $sr(j)+i*si(j)$ is applied to the isolated diagonal block in rows and columns *ktop* through *kbot*.

z If *wantz* is .TRUE., *z* is updated with transformations applied only to the submatrix $z(iloz:ihiz,kbot:ktop)$.

work(1) On exit, if *info* = 0, *work(1)* returns the optimal *lwork*.

iwork(1) On exit, if *info* = 0, *iwork(1)* returns the optimal *lwork*.

p?laqsy

Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ.

Syntax

Fortran:

```
call pslaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pdlaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pclaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
call pzlaqsy(uplo, n, a, ia, ja, desca, sr, sc, scond, amax, equed)
```

C:

```
void pslaqsy (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *sr , float *sc , float *scond , float *amax , char *equed );
void pdlaqsy (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *sr , double *sc , double *scond , double *amax , char
*equed );
void pclaqsy (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , float *sr , float *sc , float *scond , float *amax , char *equed );
void pzlaqsy (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *sr , double *sc , double *scond , double *amax , char
*equed );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?laqsy` routine equilibrates a symmetric distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the scaling factors in the vectors `sr` and `sc`. The scaling factors are computed by `p?poequ`.

Input Parameters

<code>uplo</code>	(global) CHARACTER. Specifies the upper or lower triangular part of the symmetric distributed matrix $\text{sub}(A)$ is to be referenced: = 'U': Upper triangular part; = 'L': Lower triangular part.
<code>n</code>	(global) INTEGER. The order of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.
<code>a</code>	(local). REAL for <code>pslaqsy</code> DOUBLE PRECISION for <code>pdlaqsy</code> COMPLEX for <code>pclaqsy</code> COMPLEX*16 for <code>pzlaqsy</code> . Pointer into the local memory to an array of size ($\text{lld}_a, \text{LOCc}(ja+n-1)$). On entry, this array contains the local pieces of the distributed matrix $\text{sub}(A)$. On entry, the local pieces of the distributed symmetric matrix $\text{sub}(A)$. If <code>uplo</code> = 'U', the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and the strictly lower triangular part of $\text{sub}(A)$ is not referenced. If <code>uplo</code> = 'L', the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the global array A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<code>desca</code>	(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix A .
<code>sr</code>	(local) REAL for <code>pslaqsy</code> DOUBLE PRECISION for <code>pdlaqsy</code> COMPLEX for <code>pclaqsy</code> COMPLEX*16 for <code>pzlaqsy</code> . Array, DIMENSION $\text{LOCr}(m_a)$. The scale factors for $A(ia:ia+m-1, ja:ja+n-1)$. <code>sr</code> is aligned with the distributed matrix A , and replicated across every process column. <code>sr</code> is tied to the distributed matrix A .
<code>sc</code>	(local)

```

REAL for pslaqsy
DOUBLE PRECISION for pdlaqsy
COMPLEX for pclaqsy
COMPLEX*16 for pzlaqsy.

Array, DIMENSION LOCc(m_a). The scale factors for A (ia:ia+m-1, ja:ja+n-1). sr is aligned with the distributed matrix A, and replicated across every process column. sr is tied to the distributed matrix A.

scond
(global). REAL for pslaqsy
DOUBLE PRECISION for pdlaqsy
COMPLEX for pclaqsy
COMPLEX*16 for pzlaqsy.

Ratio of the smallest sr(i) (respectively sc(j)) to the largest sr(i) (respectively sc(j)), with ia ≤ i ≤ ia+n-1 and ja ≤ j ≤ ja+n-1.

amax
(global).

REAL for pslaqsy
DOUBLE PRECISION for pdlaqsy
COMPLEX for pclaqsy
COMPLEX*16 for pzlaqsy.

Absolute value of largest distributed submatrix entry.

```

Output Parameters

a

On exit,

if *equed* = 'Y', the equilibrated matrix:

$$\text{diag}(sr(ia:ia+n-1)) * \text{sub}(A) * \text{diag}(sc(ja:ja+n-1)).$$

equed

(global) CHARACTER*1.

Specifies whether or not equilibration was done.

= 'N': No equilibration.

= 'Y': Equilibration was done, that is, sub(A) has been replaced by:

$$\text{diag}(sr(ia:ia+n-1)) * \text{sub}(A) * \text{diag}(sc(ja:ja+n-1)).$$

p?lared1d

Redistributes an array assuming that the input array, bycol, is distributed across rows and that all process columns contain the same copy of bycol.

Syntax

Fortran:

```

call pslared1d(n, ia, ja, desc, bycol, byall, work, lwork)
call pdlared1d(n, ia, ja, desc, bycol, byall, work, lwork)

```

C:

```
void pslared1d (MKL_INT *n , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc , float *bycol ,
float *byall , float *work , MKL_INT *lwork );
void pdlared1d (MKL_INT *n , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc , double
*bycol , double *byall , double *work , MKL_INT *lwork );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?lared1d` routine redistributes a 1D array. It assumes that the input array `bycol` is distributed across rows and that all process column contain the same copy of `bycol`. The output array `byall` is identical on all processes and contains the entire array.

Input Parameters

`np` = Number of local rows in `bycol()`

`n` (global) INTEGER.

The number of elements to be redistributed. $n \geq 0$.

`ia, ja` (global) INTEGER. `ia, ja` must be equal to 1.

`desc` (local) INTEGER array, size 9. A 2D array descriptor, which describes `bycol`.

`bycol` (local).

REAL for `pslared1d`

DOUBLE PRECISION for `pdlared1d`

COMPLEX for `pclared1d`

COMPLEX*16 for `pzlated1d`.

Distributed block cyclic array global size (`n`), local size `np`. `bycol` is distributed across the process rows. All process columns are assumed to contain the same value.

`work` (local).

REAL for `pslared1d`

DOUBLE PRECISION for `pdlared1d`

COMPLEX for `pclared1d`

COMPLEX*16 for `pzlated1d`.

size (`lwork`). Used to hold the buffers sent from one process to another.

`lwork` (local)

INTEGER. The size of the `work` array. $lwork \geq \text{numroc}(n, \text{desc}(\text{nb}), 0, 0, npcol)$.

Output Parameters

`byall` (global). REAL for `pslared1d`

DOUBLE PRECISION for pdlared1d
 COMPLEX for pclared1d
 COMPLEX*16 for pzlated1d.

Global size (n), local size (n). $byall$ is exactly duplicated on all processes. It contains the same values as $bycol$, but it is replicated across all processes rather than being distributed.

p?lared2d

Redistributes an array assuming that the input array byrow is distributed across columns and that all process rows contain the same copy of byrow.

Syntax

Fortran:

```
call pslared2d(n, ia, ja, desc, byrow, byall, work, lwork)
call pdlared2d(n, ia, ja, desc, byrow, byall, work, lwork)
```

C:

```
void pslared2d (MKL_INT *n , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc , float *byrow ,
float *byall , float *work , MKL_INT *lwork );
void pdlared2d (MKL_INT *n , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc , double
*byrow , double *byall , double *work , MKL_INT *lwork );
```

Include Files

- C: mkl_scalapack.h

Description

The p?lared2d routine redistributes a 1D array. It assumes that the input array $byrow$ is distributed across columns and that all process rows contain the same copy of $byrow$. The output array $byall$ will be identical on all processes and will contain the entire array.

Input Parameters

np = Number of local rows in $byrow()$

n (global) INTEGER.
 The number of elements to be redistributed. $n \geq 0$.

ia, ja (global) INTEGER. ia, ja must be equal to 1.

$desc$ (local) INTEGER array, size ($dlen_$). A 2D array descriptor, which describes $byrow$.

$byrow$ (local).
 REAL for pslared2d
 DOUBLE PRECISION for pdlared2d
 COMPLEX for pclared2d
 COMPLEX*16 for pzlated2d.

Distributed block cyclic array global size (n), local size np . $bycol$ is distributed across the process columns. All process rows are assumed to contain the same value.

<i>work</i>	(local).
	REAL for pslared2d
	DOUBLE PRECISION for pdlared2d
	COMPLEX for pclared2d
	COMPLEX*16 for pzlared2d.
	size (<i>lwork</i>). Used to hold the buffers sent from one process to another.
<i>lwork</i>	(local) INTEGER. The size of the <i>work</i> array. $lwork \geq \text{numroc}(n, \text{desc}(nb_), 0, 0, npcol)$.

Output Parameters

<i>byall</i>	(global). REAL for pslared2d
	DOUBLE PRECISION for pdlared2d
	COMPLEX for pclared2d
	COMPLEX*16 for pzlared2d.
	Global size(n), local size (n). <i>byall</i> is exactly duplicated on all processes. It contains the same values as <i>bycol</i> , but it is replicated across all processes rather than being distributed.

p?larf

Applies an elementary reflector to a general rectangular matrix.

Syntax

Fortran:

```
call pslarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pdlarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pclarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarf(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

C:

```
void pslarf (char *side , MKL_INT *m , MKL_INT *n , float *v , MKL_INT *iv , MKL_INT
*jv , MKL_INT *descv , MKL_INT *incv , float *tau , float *c , MKL_INT *ic , MKL_INT
*jc , MKL_INT *descc , float *work );
void pdlarf (char *side , MKL_INT *m , MKL_INT *n , double *v , MKL_INT *iv , MKL_INT
*jv , MKL_INT *descv , MKL_INT *incv , double *tau , double *c , MKL_INT *ic , MKL_INT
*jc , MKL_INT *descc , double *work );
void pclarf (char *side , MKL_INT *m , MKL_INT *n , MKL_Complex8 *v , MKL_INT *iv ,
MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex8 *tau , MKL_Complex8 *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work );
```

```
void pzlarf (char *side , MKL_INT *m , MKL_INT *n , MKL_Complex16 *v , MKL_INT *iv ,
MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex16 *tau , MKL_Complex16 *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work );
```

Include Files

- C: mkl_scalapack.h

Description

The p?larf routine applies a real/complex elementary reflector Q (or Q^T) to a real/complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau v^* v,$$

where τ is a real/complex scalar and v is a real/complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Input Parameters

<i>side</i>	(global). CHARACTER. = 'L': form $Q^* \text{sub}(C)$, = 'R': form $\text{sub}(C) * Q$, $Q = Q^T$.
<i>m</i>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<i>v</i>	(local). REAL for pslarf DOUBLE PRECISION for pdlarf COMPLEX for pclarf COMPLEX*16 for pzlarf. Pointer into the local memory to an array of size (<i>lld_v</i> ,*) containing the local pieces of the distributed vectors V representing the Householder transformation Q , $v(iv:iv+m-1, jv)$ if <i>side</i> = 'L' and <i>incv</i> = 1, $v(iv, jv:jv+m-1)$ if <i>side</i> = 'L' and <i>incv</i> = <i>m_v</i> , $v(iv:iv+n-1, jv)$ if <i>side</i> = 'R' and <i>incv</i> = 1, $v(iv, jv:jv+n-1)$ if <i>side</i> = 'R' and <i>incv</i> = <i>m_v</i> . The vector v is the representation of Q . v is not used if $\tau = 0$.
<i>iv</i> , <i>jv</i>	(global) INTEGER. The row and column indices in the global array V indicating the first row and the first column of the submatrix $\text{sub}(V)$, respectively.

<i>descv</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix V.
<i>incv</i>	(global) INTEGER. The global increment for the elements of <i>v</i> . Only two values of <i>incv</i> are supported in this version, namely 1 and <i>m_v</i> . <i>incv</i> must not be zero.
<i>tau</i>	(local). REAL for pslarf DOUBLE PRECISION for pdlarf COMPLEX for pclarf COMPLEX*16 for pzclarf. Array, size <i>LOCc(jv)</i> if <i>incv</i> = 1, and <i>LOCr(iv)</i> otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix <i>v</i> .
<i>c</i>	(local). REAL for pslarf DOUBLE PRECISION for pdlarf COMPLEX for pclarf COMPLEX*16 for pzclarf. Pointer into the local memory to an array of size (<i>lld_c</i> , <i>LOCc(jc+n-1)</i>), containing the local pieces of sub(<i>C</i>).
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix sub(<i>C</i>), respectively.
<i>descC</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix C.
<i>work</i>	(local). REAL for pslarf DOUBLE PRECISION for pdlarf COMPLEX for pclarf COMPLEX*16 for pzclarf. Array, size (<i>lwork</i>). If <i>incv</i> = 1, if <i>side</i> = 'L', if <i>ivcol</i> = <i>iccol</i> , <i>lwork</i> ≥ <i>nqc0</i> else

```

lwork $\geq$ mpc0 + max( 1, nqc0 )
end if
else if side = 'R',
lwork $\geq$ nqc0 + max( max( 1, mpc0 ), numroc(numroc( n+
icoffc,nb_v,0,0,npcol),nb_v,0,0,lcmq ) )
end if
else if incv = m_v,
if side = 'L',
lwork $\geq$ mpc0 + max( max( 1, nqc0 ), numroc(
numroc(m+iroffc,mb_v,0,0,nprow ),mb_v,0,0, lcmp ) )
else if side = 'R',
if ivrow = icrow,
lwork $\geq$ mpc0
else
lwork $\geq$ nqc0 + max( 1, mpc0 )
end if
end if
end if,
where lcm is the least common multiple of nprow and npcol and lcm =
ilcm( nprow, npcol ), lcmp = lcm/nprow, lcmq = lcm/npcol,
iroffc = mod( ic-1, mb_c ), icoffc = mod( jc-1, nb_c ),
icrow = indxg2p( ic, mb_c, myrow, rsrc_c, nprow ),
iccol = indxg2p( jc, nb_c, mycol, csrc_c, npcol ),
mpc0 = numroc( m+iroffc, mb_c, myrow, icrow, nprow ),
nqc0 = numroc( n+icoffc, nb_c, mycol, iccol, npcol ),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions; myrow, mycol,
nprow, and npcol can be determined by calling the subroutine
blacs_gridinfo.

```

Output Parameters

C

(local).

On exit, sub(*C*) is overwritten by the $Q^* \text{sub}(C)$ if *side* = 'L', or $\text{sub}(C) * Q$ if *side* = 'R'.

p?larfb

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

Fortran:

```
call pslarfb(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c, ic, jc,
descc, work)

call pdlarfb(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c, ic, jc,
descc, work)

call pclarf(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c, ic, jc,
descc, work)

call pzlarfb(side, trans, direct, storev, m, n, k, v, iv, jv, descv, t, c, ic, jc,
descc, work)
```

C:

```
void pslarfb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , float *v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv ,
float *t , float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , float *work );

void pdlarfb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , double *v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv ,
double *t , double *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , double *work );

void pclarf (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_Complex8 *v , MKL_INT *iv , MKL_INT *jv , MKL_INT
*descv , MKL_Complex8 *t , MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT
*descc , MKL_Complex8 *work );

void pzlarfb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_Complex16 *v , MKL_INT *iv , MKL_INT *jv , MKL_INT
*descv , MKL_Complex16 *t , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT
*descc , MKL_Complex16 *work );
```

Include Files

- C: mkl_scalapack.h

Description

The p?larfb routine applies a real/complex block reflector Q or its transpose Q^T /conjugate transpose Q^H to a real/complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ from the left or the right.

Input Parameters

<i>side</i>	(global) CHARACTER.
	if <i>side</i> = 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the Left;
	if <i>side</i> = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the Right.
<i>trans</i>	(global) CHARACTER.
	if <i>trans</i> = 'N': no transpose, apply Q ;
	for real flavors, if <i>trans</i> ='T': transpose, apply Q^T
	for complex flavors, if <i>trans</i> = 'C': conjugate transpose, apply Q^H ;
<i>direct</i>	(global) CHARACTER. Indicates how Q is formed from a product of elementary reflectors.
	if <i>direct</i> = 'F': $Q = H(1)*H(2)*\dots*H(k)$ (Forward)

if *direct* = 'B': $Q = H(k) * \dots * H(2) * H(1)$ (Backward)
storev
 (global) CHARACTER.
 Indicates how the vectors that define the elementary reflectors are stored:
if *storev* = 'C': Columnwise
if *storev* = 'R': Rowwise.

m
 (global) INTEGER.
 The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. ($m \geq 0$).

n
 (global) INTEGER.
 The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. ($n \geq 0$).

k
 (global) INTEGER.
 The order of the matrix T .

v
 (local).
 REAL for `pstrarfb`
 DOUBLE PRECISION for `pdstrarfb`
 COMPLEX for `pclarfb`
 COMPLEX*16 for `pzclarfb`.
 Pointer into the local memory to an array of size
 $(\text{lld}_v, \text{LOC}_c(jv+k-1))$ if *storev* = 'C',
 $(\text{lld}_v, \text{LOC}_c(jv+m-1))$ if *storev* = 'R' and *side* = 'L',
 $(\text{lld}_v, \text{LOC}_c(jv+n-1))$ if *storev* = 'R' and *side* = 'R'.
 It contains the local pieces of the distributed vectors V representing the Householder transformation.
if *storev* = 'C' and *side* = 'L', $\text{lld}_v \geq \max(1, \text{LOC}_r(iv+m-1))$;
if *storev* = 'C' and *side* = 'R', $\text{lld}_v \geq \max(1, \text{LOC}_r(iv+n-1))$;
if *storev* = 'R', $\text{lld}_v \geq \text{LOC}_r(jv+k-1)$.

iv, jv
 (global) INTEGER.
 The row and column indices in the global array V indicating the first row and the first column of the submatrix $\text{sub}(V)$, respectively.

descv
 (global and local) INTEGER array, size (*dlen_*). The array descriptor for the distributed matrix V .

c
 (local).
 REAL for `pstrarfb`
 DOUBLE PRECISION for `pdstrarfb`
 COMPLEX for `pclarfb`

`COMPLEX*16` **for** `pzlarfb`.

Pointer into the local memory to an array of size (`lld_c`, `LOCc(jc+n-1)`), containing the local pieces of $\text{sub}(C)$.

`ic, jc`

(global) `INTEGER`. The row and column indices in the global array C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.

`descC`

(global and local) `INTEGER` array, size (`dlen_`). The array descriptor for the distributed matrix C .

`work`

(local).

`REAL` **for** `pslarfb`

`DOUBLE PRECISION` **for** `pdlarfb`

`COMPLEX` **for** `pclarfb`

`COMPLEX*16` **for** `pzlarfb`.

Workspace array, size (`lwork`).

If `storev = 'C'`,

if `side = 'L'`,

$lwork \geq (nqc0 + mpc0) * k$

else if `side = 'R'`,

$lwork \geq (nqc0 + \max(npv0 + \text{numroc}(\text{numroc}(n + ioffc, nb_v, 0, 0, npcol), nb_v, 0, 0, lcmq), mpc0)) * k$

end if

else if `storev = 'R'`,

if `side = 'L'`,

$lwork \geq (mpc0 + \max(mqv0 + \text{numroc}(\text{numroc}(m + iroffc, mb_v, 0, 0, nprow), mb_v, 0, 0, icmp), nqc0)) * k$

else if `side = 'R'`,

$lwork \geq (mpc0 + nqc0) * k$

end if

end if,

where

$lcmq = lcm / npcol$ with $lcm = \text{iclm}(nprow, npcol)$,

$iroffv = \text{mod}(iv-1, mb_v)$, $icoffv = \text{mod}(jv-1, nb_v)$,

$ivrow = \text{indxg2p}(iv, mb_v, myrow, rsrc_v, nprow)$,

$ivcol = \text{indxg2p}(jv, nb_v, mycol, csrc_v, npcol)$,

$MqV0 = \text{numroc}(m+icoffv, nb_v, mycol, ivcol, npcol)$,

$NpV0 = \text{numroc}(n+iroffv, mb_v, myrow, ivrow, nprow)$,

```

iroffc = mod( ic-1, mb_c ), icoffc = mod( jc-1, nb_c ),
icrow = indxg2p( ic, mb_c, myrow, rsrc_c, nprov ),
iccol = indxg2p( jc, nb_c, mycol, csrc_c, npcol ),
MpC0 = numroc( m+iroffc, mb_c, myrow, icrow, nprov ),
NpC0 = numroc( n+icoffc, mb_c, myrow, icrow, nprov ),
NqC0 = numroc( n+icoffc, nb_c, mycol, iccol, npcol ),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions; myrow, mycol,
nprov, and npcol can be determined by calling the subroutine
blacs_gridinfo.
```

Output Parameters

- t* (local).
 REAL for pslarfb
 DOUBLE PRECISION for pdlarfb
 COMPLEX for pclarfb
 COMPLEX*16 for pzlarfb.
 Array, size(*mb_v*, *mb_v*) if *storev* = 'R', and (*nb_v*, *nb_v*) if
 storev = 'C'. The triangular matrix *t* is the representation of the block
 reflector.
- c* (local).
 On exit, *sub(C)* is overwritten by the $Q^* \text{sub}(C)$, or $Q'^* \text{sub}(C)$, or $\text{sub}(C) * Q$, or $\text{sub}(C) * Q'$. *Q'* is transpose (conjugate transpose) of *Q*.

p?larfc

Applies the conjugate transpose of an elementary reflector to a general matrix.

Syntax

Fortran:

```
call pclarfc(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarfc(side, m, n, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

C:

```
void pclarfc (char *side , MKL_INT *m , MKL_INT *n , MKL_Complex8 *v , MKL_INT *iv ,
MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex8 *tau , MKL_Complex8 *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work );
void pzlarfc (char *side , MKL_INT *m , MKL_INT *n , MKL_Complex16 *v , MKL_INT *iv ,
MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex16 *tau , MKL_Complex16 *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?larfc` routine applies a complex elementary reflector Q^H to a complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = i - \tau v^* v^* v,$$

where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Input Parameters

<code>side</code>	(global) CHARACTER. if <code>side</code> = 'L': form $Q^H * \text{sub}(C)$; if <code>side</code> = 'R': form $\text{sub}(C) * Q^H$.
<code>m</code>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. ($n \geq 0$).
<code>v</code>	(local). COMPLEX for <code>pclarfc</code> COMPLEX*16 for <code>pzlarfc</code> . Pointer into the local memory to an array of size (<code>lld_v,*</code>) containing the local pieces of the distributed vectors v representing the Householder transformation Q , <code>v(iv:iv+m-1, jv)</code> if <code>side</code> = 'L' and <code>incv</code> = 1, <code>v(iv, jv:jv+m-1)</code> if <code>side</code> = 'L' and <code>incv</code> = <code>m_v</code> , <code>v(iv:iv+n-1, jv)</code> if <code>side</code> = 'R' and <code>incv</code> = 1, <code>v(iv, jv:jv+n-1)</code> if <code>side</code> = 'R' and <code>incv</code> = <code>m_v</code> . The vector v is the representation of Q . v is not used if $\tau = 0$.
<code>iv, jv</code>	(global) INTEGER. The row and column indices in the global array V indicating the first row and the first column of the submatrix $\text{sub}(V)$, respectively.
<code>descv</code>	(global and local) INTEGER array, size (<code>dlen_</code>). The array descriptor for the distributed matrix V .
<code>incv</code>	(global) INTEGER. The global increment for the elements of v . Only two values of <code>incv</code> are supported in this version, namely 1 and <code>m_v</code> . <code>incv</code> must not be zero.
<code>tau</code>	(local)

COMPLEX for `pclarfc`
 COMPLEX*16 for `pzlarfc`.
 Array, size $LOCc(jv)$ if $incv = 1$, and $LOCr(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors.
 τ is tied to the distributed matrix V .

c (local).

COMPLEX for `pclarfc`
 COMPLEX*16 for `pzlarfc`.

Pointer into the local memory to an array of size (lld_c , $LOCc(jc+n-1)$), containing the local pieces of $\text{sub}(C)$.

ic, jc (global) INTEGER.

The row and column indices in the global array C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.

$desc_c$ (global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix C .

$work$ (local).

COMPLEX for `pclarfc`
 COMPLEX*16 for `pzlarfc`.

Workspace array, size ($lwork$).

```
If  $incv = 1$ ,
  if  $side = 'L'$ ,
    if  $ivcol = iccol$ ,
       $lwork \geq nqc0$ 
    else
       $lwork \geq mpc0 + \max( 1, nqc0 )$ 
    end if
  else if  $side = 'R'$ ,
     $lwork \geq nqc0 + \max( \max( 1, mpc0 ), \text{numroc}( \text{numroc}($ 
       $n+icoffc, nb\_v, 0, 0, npcol ), nb\_v, 0, 0, lcmq ) )$ 
  end if
  else if  $incv = m\_v$ ,
    if  $side = 'L'$ ,
       $lwork \geq mpc0 + \max( \max( 1, nqc0 ), \text{numroc}( \text{numroc}($ 
         $m+iroffc, mb\_v, 0, 0, nprow ), mb\_v, 0, 0, lcmq ) )$ 
    else if  $side = 'R'$ ,
      if  $ivrow = icrow$ ,
         $lwork \geq mpc0$ 
      else
```

```

lwork  $\geq nqc0 + \max( 1, mpc0 )$ 
end if
end if
end if,
where lcm is the least common multiple of nprow and npcol and lcm =
ilcm(nprow, npcol),
lcmp = lcm/nprow, lcmq = lcm/npcol,
iroffc = mod(ic-1, mb_c), icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow),
iccol = indxg2p(jc, nb_c, mycol, csrc_c, npcol),
mpc0 = numroc(m+iroffc, mb_c, myrow, icrow, nprow),
nqc0 = numroc(n+icoffc, nb_c, mycol, iccol, npcol),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions; myrow, mycol,
nprow, and npcol can be determined by calling the subroutine
blacs_gridinfo.
```

Output Parameters

c

(local).

On exit, sub(*C*) is overwritten by the $Q^H * \text{sub}(C)$ if side = 'L', or $\text{sub}(C) * Q^H$ if side = 'R'.

p?larfg

Generates an elementary reflector (*Householder matrix*).

Syntax

Fortran:

```
call pslarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pdlarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pclarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
call pzlarfg(n, alpha, iax, jax, x, ix, jx, descx, incx, tau)
```

C:

```
void pslarfg (MKL_INT *n , float *alpha , MKL_INT *iax , MKL_INT *jax , float *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , float *tau );
void pdlarfg (MKL_INT *n , double *alpha , MKL_INT *iax , MKL_INT *jax , double *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , double *tau );
void pclarfg (MKL_INT *n , MKL_Complex8 *alpha , MKL_INT *iax , MKL_INT *jax ,
MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx ,
MKL_Complex8 *tau );
```

```
void pzlarfg (MKL_INT *n , MKL_Complex16 *alpha , MKL_INT *iax , MKL_INT *jax ,
MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT *incx ,
MKL_Complex16 *tau );
```

Include Files

- C: mkl_scalapack.h

Description

The p?larfg routine generates a real/complex elementary reflector H of order n , such that

$$H^* \text{sub}(X) = H^* \begin{pmatrix} x(iax, jax) \\ x \end{pmatrix} = \begin{pmatrix} alpha \\ 0 \end{pmatrix}, \quad H^* H = I,$$

where α is a scalar (a real scalar - for complex flavors), and $\text{sub}(X)$ is an $(n-1)$ -element real/complex distributed vector $X(ix:ix+n-2, jx)$ if $incx = 1$ and $X(ix, jx:jx+n-2)$ if $incx = descx(m_)$. H is represented in the form

$$H = I - tau * \begin{pmatrix} 1 \\ v \end{pmatrix} * (1 v')$$

where τ is a real/complex scalar and v is a real/complex $(n-1)$ -element vector. Note that H is not Hermitian.

If the elements of $\text{sub}(X)$ are all zero (and $X(iax, jax)$ is real for complex flavors), then $\tau = 0$ and H is taken to be the unit matrix.

Otherwise $1 \leq \text{real}(\tau) \leq 2$ and $\text{abs}(\tau-1) \leq 1$.

Input Parameters

n	(global) INTEGER. The global order of the elementary reflector. $n \geq 0$.
iax, jax	(global) INTEGER. The global row and column indices in x of $X(iax, jax)$.
x	(local). REAL for pslarfg DOUBLE PRECISION for pdlarfg COMPLEX for pclarfg COMPLEX*16 for pzlarfg. Pointer into the local memory to an array of size ($lld_x, *$). This array contains the local pieces of the distributed vector $\text{sub}(X)$. Before entry, the incremented array $\text{sub}(X)$ must contain vector x .
ix, jx	(global) INTEGER.

The row and column indices in the global array X indicating the first row and the first column of $\text{sub}(X)$, respectively.

<i>descx</i>	(global and local) INTEGER. Array of size (<i>dlen_</i>). The array descriptor for the distributed matrix X .
<i>incx</i>	(global) INTEGER. The global increment for the elements of x . Only two values of <i>incx</i> are supported in this version, namely 1 and m_x . <i>incx</i> must not be zero.

Output Parameters

<i>alpha</i>	(local) REAL for pslafg DOUBLE PRECISION for pdlafg COMPLEX for pclafg COMPLEX*16 for pzlafg. On exit, <i>alpha</i> is computed in the process scope having the vector $\text{sub}(X)$.
<i>x</i>	(local). On exit, it is overwritten with the vector v .
<i>tau</i>	(local). REAL for pslarfg DOUBLE PRECISION for pdlarfg COMPLEX for pclarfg COMPLEX*16 for pzlarfg. Array, size $LOCc(jx)$ if <i>incx</i> = 1, and $LOCr(ix)$ otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix X .

p?larft

Forms the triangular vector T of a block reflector $H = I - V * T * V^H$.

Syntax

Fortran:

```
call pslarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pdlarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pclarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pzlarft(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
```

C:

```
void pslarft (char *direct , char *storev , MKL_INT *n , MKL_INT *k , float *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , float *tau , float *t , float *work );
```

```

void pdlarft (char *direct , char *storev , MKL_INT *n , MKL_INT *k , double *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , double *tau , double *t , double *work );
void pclarft (char *direct , char *storev , MKL_INT *n , MKL_INT *k , MKL_Complex8 *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_Complex8 *tau , MKL_Complex8 *t ,
MKL_Complex8 *work );
void pzlarft (char *direct , char *storev , MKL_INT *n , MKL_INT *k , MKL_Complex16
*v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_Complex16 *tau , MKL_Complex16
*t , MKL_Complex16 *work );

```

Include Files

- C: mkl_scalapack.h

Description

The p?larft routine forms the triangular factor T of a real/complex block reflector H of order n , which is defined as a product of k elementary reflectors.

If $\text{direct} = \text{'F'}$, $H = H(1) * H(2) \dots * H(k)$, and T is upper triangular;

If $\text{direct} = \text{'B'}$, $H = H(k) * \dots * H(2) * H(1)$, and T is lower triangular.

If $\text{storev} = \text{'C'}$, the vector which defines the elementary reflector $H(i)$ is stored in the i -th column of the distributed matrix V , and

$$H = I - V^* T^* V$$

If $\text{storev} = \text{'R'}$, the vector which defines the elementary reflector $H(i)$ is stored in the i -th row of the distributed matrix V , and

$$H = I - V'^* T^* V.$$

Input Parameters

direct

(global) CHARACTER*1.

Specifies the order in which the elementary reflectors are multiplied to form the block reflector:

if $\text{direct} = \text{'F'}$: $H = H(1) * H(2) * \dots * H(k)$ (forward)

if $\text{direct} = \text{'B'}$: $H = H(k) * \dots * H(2) * H(1)$ (backward).

storev

(global) CHARACTER*1.

Specifies how the vectors that define the elementary reflectors are stored (See Application Notes below):

if $\text{storev} = \text{'C'}$: columnwise;

if $\text{storev} = \text{'R'}$: rowwise.

n

(global) INTEGER.

The order of the block reflector H . $n \geq 0$.

k

(global) INTEGER.

The order of the triangular factor T , is equal to the number of elementary reflectors.

$$1 \leq k \leq mb_v (= nb_v).$$

<i>v</i>	REAL for pslarft DOUBLE PRECISION for pdlarft COMPLEX for pclarft COMPLEX*16 for pzlarft. Pointer into the local memory to an array of local size $(LOCr(iv+n-1), LOCc(jv+k-1))$ if <i>storev</i> = 'C', and $(LOCr(iv+k-1), LOCc(jv+n-1))$ if <i>storev</i> = 'R'. The distributed matrix <i>V</i> contains the Householder vectors. (See <i>Application Notes</i> below).
<i>iv</i> , <i>jv</i>	(global) INTEGER. The row and column indices in the global array <i>v</i> indicating the first row and the first column of the submatrix sub(<i>V</i>), respectively.
<i>descv</i>	(local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>V</i> .
<i>tau</i>	(local) REAL for pslarft DOUBLE PRECISION for pdlarft COMPLEX for pclarft COMPLEX*16 for pzlarft. Array, size $LOCr(iv+k-1)$ if <i>incv</i> = <i>m_v</i> , and $LOCc(jv+k-1)$ otherwise. This array contains the Householder scalars related to the Householder vectors. <i>tau</i> is tied to the distributed matrix <i>V</i> .
<i>work</i>	(local). REAL for pslarft DOUBLE PRECISION for pdlarft COMPLEX for pclarft COMPLEX*16 for pzlarft. Workspace array, size $(k*(k-1)/2)$.

Output Parameters

<i>v</i>	REAL for pslarft DOUBLE PRECISION for pdlarft COMPLEX for pclarft COMPLEX*16 for pzlarft.
<i>t</i>	(local) REAL for pslarft

DOUBLE PRECISION for pdlarft

COMPLEX for pclarft

COMPLEX*16 for pzlarft.

Array, size nb_v by nb_v if $storev = 'C'$, and (mb_v, mb_v) otherwise. It contains the k -by- k triangular factor of the block reflector associated with v . If $direct = 'F'$, t is upper triangular;

if $direct = 'B'$, t is lower triangular.

Application Notes

The shape of the matrix V and the storage of the vectors that define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

$\begin{matrix} \text{v1} & \text{v2v} & \text{v3} \\ \text{v1} & \text{v2} & \text{v3} \\ 1 & \text{v2} & \text{v3} \\ 1 & \text{v3} \\ 1 \end{matrix}$

$\begin{matrix} \text{v1} & \text{v2v} & \text{v3} \\ \text{v1} & \text{v2} & \text{v3} \\ 1 & \text{v2} & \text{v3} \\ 1 & \text{v3} \\ 1 \end{matrix}$

$$\begin{aligned} & \text{direct} = 'B' \text{ and } storev = 'C' \\ & V(iv : iv + n - 1, \\ & \quad jv : jv + k - 1) = \begin{bmatrix} v1 & v2v & v3 \\ v1 & v2 & v3 \\ 1 & v2 & v3 \\ 1 & v3 \\ 1 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} & \text{direct} = 'B' \text{ and } storev = 'R' \\ & V(iv : iv + k - 1, \\ & \quad jv : jv + n - 1) = \begin{bmatrix} v1 & v1 & 1 \\ v2 & v2 & v2 & 1 \\ v3 & v3 & v3 & v3 & 1 \end{bmatrix} \end{aligned}$$

p?larz

Applies an elementary reflector as returned by p?tzrzf to a general matrix.

Syntax

Fortran:

```
call pslarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pdlarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pclarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarz(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

C:

```
void pslarz (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , float *v , MKL_INT
*iv , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , float *tau , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , float *work );
```

```

void pdlarz (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , double *v , MKL_INT
*iw , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work );

void pclarz (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex8 *v ,
MKL_INT *iw , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work );

void pzlarz (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex16 *v ,
MKL_INT *iw , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work );

```

Include Files

- C: mkl_scalapack.h

Description

The `p?larz` routine applies a real/complex elementary reflector Q (or Q^T) to a real/complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = I - \tau v v^*,$$

where τ is a real/complex scalar and v is a real/complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Q is a product of k elementary reflectors as returned by [p?tzrzf](#).

Input Parameters

`side`

(global) CHARACTER.

if `side` = 'L': form $Q^* \text{sub}(C)$,

if `side` = 'R': form $\text{sub}(C)^* Q$, $Q = Q^T$ (for real flavors).

`m`

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. ($m \geq 0$).

`n`

(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. ($n \geq 0$).

`l`

(global) INTEGER.

The columns of the distributed submatrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. If `side` = 'L', $m \geq l \geq 0$,

if `side` = 'R', $n \geq l \geq 0$.

`v`

(local).

REAL for `pslarz`

DOUBLE PRECISION for `pdlarz`

COMPLEX for `pclarz`

COMPLEX*16 for `pzlarz`.

Pointer into the local memory to an array of size ($lld_v, *$) containing the local pieces of the distributed vectors v representing the Householder transformation Q ,

```
v(iv:iv+l-1, jv) if side = 'L' and incv = 1,  
v(iv, jv:jv+l-1) if side = 'L' and incv = m_v,  
v(iv:iv+l-1, jv) if side = 'R' and incv = 1,  
v(iv, jv:jv+l-1) if side = 'R' and incv = m_v.
```

The vector v in the representation of Q . v is not used if $tau = 0$.

iv, jv

(global) INTEGER. The row and column indices in the global array V indicating the first row and the first column of the submatrix $\text{sub}(V)$, respectively.

descv

(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix V .

incv

(global) INTEGER.

The global increment for the elements of v . Only two values of $incv$ are supported in this version, namely 1 and m_v .

$incv$ must not be zero.

tau

(local)

```
REAL for pslarz  
DOUBLE PRECISION for pdlarz  
COMPLEX for pclarz  
COMPLEX*16 for pzlarz.
```

Array, size $LOCc(jv)$ if $incv = 1$, and $LOCr(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors. tau is tied to the distributed matrix V .

c

(local).

```
REAL for pslarz  
DOUBLE PRECISION for pdlarz  
COMPLEX for pclarz  
COMPLEX*16 for pzlarz.
```

Pointer into the local memory to an array of size ($lld_c, LOCc(jc+n-1)$), containing the local pieces of $\text{sub}(C)$.

ic, jc

(global) INTEGER.

The row and column indices in the global array C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.

descC

(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix C .

work

(local).

```

REAL for pslarz
DOUBLE PRECISION for pdlarz
COMPLEX for pclarz
COMPLEX*16 for pzlarz.

Array, size (lwork)

If incv = 1,
  if side = 'L' ,
    if ivcol = iccol,
      lwork  $\geq NqC0$ 
    else
      lwork  $\geq MpC0 + \max(1, NqC0)$ 
    end if
  else if side = 'R' ,
    lwork  $\geq NqC0 + \max(\max(1, MpC0), \text{numroc}(\text{numroc}(n+icoffc, nb\_v, 0, 0, npcol), nb\_v, 0, 0, lcmq)))$ 
  end if
  else if incv = m_v,
    if side = 'L' ,
      lwork  $\geq MpC0 + \max(\max(1, NqC0), \text{numroc}(\text{numroc}(m+iroffc, mb\_v, 0, 0, nprow), mb\_v, 0, 0, lcm))$ 
    else if side = 'R' ,
      if ivrow = icrow,
        lwork  $\geq MpC0$ 
      else
        lwork  $\geq NqC0 + \max(1, MpC0)$ 
      end if
    end if
  end if.

```

Here lcm is the least common multiple of $nprow$ and $npcol$ and

$$\begin{aligned} lcm &= \text{ilcm}(nprow, npcol), lcm = lcm / nprow, \\ lcmq &= lcm / npcol, \\ iroffc &= \text{mod}(ic-1, mb_c), icoffc = \text{mod}(jc-1, nb_c), \\ icrow &= \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprow), \\ iccol &= \text{indxg2p}(jc, nb_c, mycol, csrc_c, npcol), \\ mpco &= \text{numroc}(m+iroffc, mb_c, myrow, icrow, nprow), \\ nqco &= \text{numroc}(n+icoffc, nb_c, mycol, iccol, npcol), \end{aligned}$$

`ilcm`, `indxg2p`, and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprov`, and `ncol` can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

`c` (local).

On exit, `sub(C)` is overwritten by the $Q^* \text{sub}(C)$ if `side = 'L'`, or `sub(C) * Q` if `side = 'R'`.

p?larzb

Applies a block reflector or its transpose/conjugate-transpose as returned by p?tzrzf to a general matrix.

Syntax

Fortran:

```
call pslarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc,
descc, work)

call pdlarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc,
descc, work)

call pclarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc,
descc, work)

call pzlarzb(side, trans, direct, storev, m, n, k, l, v, iv, jv, descv, t, c, ic, jc,
descc, work)
```

C:

```
void pslarzb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_INT *l , float *v , MKL_INT *iv , MKL_INT *jv , MKL_INT
*descv , float *t , float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , float
*work );

void pdlarzb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_INT *l , double *v , MKL_INT *iv , MKL_INT *jv , MKL_INT
*descv , double *t , double *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , double
*work );

void pclarzb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_INT *l , MKL_Complex8 *v , MKL_INT *iv , MKL_INT *jv ,
MKL_INT *descv , MKL_Complex8 *t , MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc , MKL_Complex8 *work );

void pzlarzb (char *side , char *trans , char *direct , char *storev , MKL_INT *m ,
MKL_INT *n , MKL_INT *k , MKL_INT *l , MKL_Complex16 *v , MKL_INT *iv , MKL_INT *jv ,
MKL_INT *descv , MKL_Complex16 *t , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc , MKL_Complex16 *work );
```

Include Files

- C: `mkl_scalapack.h`

Description

The `p?larzb` routine applies a real/complex block reflector Q or its transpose Q^T (conjugate transpose Q^H for complex flavors) to a real/complex distributed m -by- n matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$ from the left or the right.

Q is a product of k elementary reflectors as returned by `p?tzrzf`.

Currently, only `storev = 'R'` and `direct = 'B'` are supported.

Input Parameters

<code>side</code>	(global) CHARACTER. if <code>side</code> = ' <code>L</code> ': apply Q or Q^T (Q^H for complex flavors) from the Left; if <code>side</code> = ' <code>R</code> ': apply Q or Q^T (Q^H for complex flavors) from the Right.
<code>trans</code>	(global) CHARACTER. if <code>trans</code> = ' <code>N</code> ': No transpose, apply Q ; If <code>trans='T'</code> : Transpose, apply Q^T (real flavors); If <code>trans='C'</code> : Conjugate transpose, apply Q^H (complex flavors).
<code>direct</code>	(global) CHARACTER. Indicates how H is formed from a product of elementary reflectors. if <code>direct</code> = ' <code>F</code> ': $H = H(1)*H(2)*\dots*H(k)$ - forward (not supported) ; if <code>direct</code> = ' <code>B</code> ': $H = H(k)*\dots*H(2)*H(1)$ - backward.
<code>storev</code>	(global) CHARACTER. Indicates how the vectors that define the elementary reflectors are stored: if <code>storev</code> = ' <code>C</code> ': columnwise (not supported). if <code>storev</code> = ' <code>R</code> ': rowwise.
<code>m</code>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. ($n \geq 0$).
<code>k</code>	(global) INTEGER. The order of the matrix T . (= the number of elementary reflectors whose product defines the block reflector).
<code>l</code>	(global) INTEGER. The columns of the distributed submatrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. If <code>side</code> = ' <code>L</code> ', $m \geq l \geq 0$, if <code>side</code> = ' <code>R</code> ', $n \geq l \geq 0$.

<i>v</i>	(local). REAL for pslarzb DOUBLE PRECISION for pdlarzb COMPLEX for pclarzb COMPLEX*16 for pzlarzb. Pointer into the local memory to an array of size (<i>lld_v</i> , <i>LOCc(jv+m-1)</i>) if <i>side</i> = 'L', (<i>lld_v</i> , <i>LOCc(jv+m-1)</i>) if <i>side</i> = 'R'. It contains the local pieces of the distributed vectors <i>V</i> representing the Householder transformation as returned by p?tzrzf. $lld_v \geq LOCr(iv+k-1)$.
<i>iv, jv</i>	(global) INTEGER. The row and column indices in the global array <i>V</i> indicating the first row and the first column of the submatrix sub(<i>V</i>), respectively.
<i>descv</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>V</i> .
<i>t</i>	(local) REAL for pslarzb DOUBLE PRECISION for pdlarzb COMPLEX for pclarzb COMPLEX*16 for pzlarzb. Array, DIMENSION <i>mb_v</i> by <i>mb_v</i> . The lower triangular matrix <i>T</i> in the representation of the block reflector.
<i>c</i>	(local). REAL for pslarfb DOUBLE PRECISION for pdlarfb COMPLEX for pclarfb COMPLEX*16 for pzlarfb. Pointer into the local memory to an array of size (<i>lld_c</i> , <i>LOCc(jc+n-1)</i>). On entry, the <i>m</i> -by- <i>n</i> distributed matrix sub(<i>C</i>).
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the global array <i>c</i> indicating the first row and the first column of the submatrix sub(<i>C</i>), respectively.
<i>descC</i>	(global and local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>C</i> .
<i>work</i>	(local). REAL for pslarzb DOUBLE PRECISION for pdlarzb

```

COMPLEX for pclarzb
COMPLEX*16 for pzlarzb.

Array, size (lwork).

If storev = 'C' ,
    if side = 'L' ,
        lwork ≥(nqc0 + mpc0)* k
    else if side = 'R' ,
        lwork ≥ (nqc0 + max(npv0 + numroc(numroc(n+icoffc, nb_v, 0, 0,
npcol),
                                         nb_v, 0, 0, lcmq), mpc0))* k
    end if
else if storev = 'R' ,
    if side = 'L' ,
        lwork ≥ (mpc0 + max(mqv0 + numroc(numroc(m+iroffc, mb_v, 0, 0,
nprov),
                                         mb_v, 0, 0, lcmq), nqc0))* k
    else if side = 'R' ,
        lwork ≥ (mpc0 + nqc0) * k
    end if
end if.

```

Here $lcmq = lcm/npcol$ with $lcm = iclm(nprov, npcol)$,
 $iroffv = \text{mod}(iv-1, mb_v)$, $icoffv = \text{mod}(jv-1, nb_v)$,
 $ivrow = \text{indxg2p}(iv, mb_v, myrow, rsrc_v, nprov)$,
 $ivcol = \text{indxg2p}(jv, nb_v, mycol, csrc_v, npcol)$,
 $mqv0 = \text{numroc}(m+icoffv, nb_v, mycol, ivcol, npcol)$,
 $npv0 = \text{numroc}(n+iroffv, mb_v, myrow, ivrow, nprov)$,
 $iroffc = \text{mod}(ic-1, mb_c)$, $icoffc = \text{mod}(jc-1, nb_c)$,
 $icrow = \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprov)$,
 $iccol = \text{indxg2p}(jc, nb_c, mycol, csrc_c, npcol)$,
 $mpc0 = \text{numroc}(m+iroffc, mb_c, myrow, icrow, nprov)$,
 $npc0 = \text{numroc}(n+icoffc, mb_c, myrow, icrow, nprov)$,
 $nqc0 = \text{numroc}(n+icoffc, nb_c, mycol, iccol, npcol)$,
 ilcm , indxg2p , and numroc are ScaLAPACK tool functions; $myrow$, $mycol$,
 $nprov$, and $npcol$ can be determined by calling the subroutine
 blacs_gridinfo .

Output Parameters

c

(local).

On exit, $\text{sub}(C)$ is overwritten by the $Q^* \text{sub}(C)$, or $Q'^* \text{sub}(C)$, or $\text{sub}(C)^* Q$, or $\text{sub}(C)^* Q'$, where Q' is the transpose (conjugate transpose) of Q .

p?larzc

Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by p?tzrzf to a general matrix.

Syntax

Fortran:

```
call pclarzc(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
call pzlarzc(side, m, n, l, v, iv, jv, descv, incv, tau, c, ic, jc, descc, work)
```

C:

```
void pclarzc (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex8 *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work );
void pzlarzc (char *side , MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex16 *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_INT *incv , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work );
```

Include Files

- C: mkl_scalapack.h

Description

The p?larzc routine applies a complex elementary reflector Q^H to a complex m -by- n distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. Q is represented in the form

$$Q = i - \tau u^* v^* v,$$

where τ is a complex scalar and v is a complex vector.

If $\tau = 0$, then Q is taken to be the unit matrix.

Q is a product of k elementary reflectors as returned by p?tzrzf.

Input Parameters

side

(global) CHARACTER.

```
if side = 'L': form  $Q^H * \text{sub}(C)$ ;
if side = 'R': form  $\text{sub}(C) * Q^H$ .
```

m

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. ($m \geq 0$).

n

(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. ($n \geq 0$).

l

(global) INTEGER.

The columns of the distributed submatrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors.

If $\text{side} = \text{'L'}$, $m \geq l \geq 0$,

if $\text{side} = \text{'R'}$, $n \geq l \geq 0$.

v

(local).

COMPLEX for `pclarzc`COMPLEX*16 for `pzlarzc`.

Pointer into the local memory to an array of size ($lld_v, *$) containing the local pieces of the distributed vectors v representing the Householder transformation Q ,

$v(iv:iv+l-1, jv)$ if $\text{side} = \text{'L'}$ and $\text{incv} = 1$,

$v(iv, jv:jv+l-1)$ if $\text{side} = \text{'L'}$ and $\text{incv} = m_v$,

$v(iv:iv+l-1, jv)$ if $\text{side} = \text{'R'}$ and $\text{incv} = 1$,

$v(iv, jv:jv+l-1)$ if $\text{side} = \text{'R'}$ and $\text{incv} = m_v$.

The vector v in the representation of Q . v is not used if $\tau_{au} = 0$.

iv, jv

(global) INTEGER.

The row and column indices in the global array V indicating the first row and the first column of the submatrix $\text{sub}(V)$, respectively.

descv

(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix V .

incv

(global) INTEGER.

The global increment for the elements of v . Only two values of incv are supported in this version, namely 1 and m_v .

incv must not be zero.

tau

(local)

COMPLEX for `pclarzc`COMPLEX*16 for `pzlarzc`.

Array, size $\text{LOCc}(jv)$ if $\text{incv} = 1$, and $\text{LOCr}(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors.

τ_{au} is tied to the distributed matrix V .

c

(local).

COMPLEX for `pclarzc`COMPLEX*16 for `pzlarzc`.

Pointer into the local memory to an array of size ($lld_c, \text{LOCc}(jc+n-1)$), containing the local pieces of $\text{sub}(C)$.

ic, jc

(global) INTEGER.

The row and column indices in the global array C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.

descC (global and local) INTEGER array, size (*dlen_*). The array descriptor for the distributed matrix C .

work (local).

```

If incv = 1,
  if side = 'L' ,
    if ivcol = iccol,
      lwork ≥ nqc0
    else
      lwork ≥ mpc0 + max(1, nqc0)
    end if
  else if side = 'R' ,
    lwork ≥ nqc0 + max(max(1, mpc0), numroc(numroc(n+icoffc, nb_v, 0, 0,
npcol),
      nb_v, 0, 0, lcmq) )
  end if
else if incv = m_v,
  if side = 'L' ,
    lwork ≥ mpc0 + max(max(1, nqc0), numroc(numroc(m+iroffc, mb_v, 0, 0,
nprow),
      mb_v, 0, 0, lcmp) )
  else if side = 'R',
    if ivrow = icrow,
      lwork ≥ mpc0
    else
      lwork ≥ nqc0 + max(1, mpc0)
    end if
  end if
end if

```

Here *lcm* is the least common multiple of *nprow* and *npcol*;

```

lcm = ilcm(nprow, npcol), lcmp = lcm/nprow, lcmq= lcm/npcol,
iroffc = mod(ic-1, mb_c), icoffc= mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow),
iccol = indxg2p(jc, nb_c, mycol, csrc_c, npcol),
mpc0 = numroc(m+iroffc, mb_c, myrow, icrow, nprow),
nqc0 = numroc(n+icoffc, nb_c, mycol, iccol, npcol),

```

`ilcm`, `indxg2p`, and `numroc` are ScaLAPACK tool functions;
`myrow`, `mycol`, `nprov`, and `npcol` can be determined by calling the subroutine `blacs_gridinfo`.

Output Parameters

`c` (local).

On exit, $\text{sub}(C)$ is overwritten by the $Q^H \text{sub}(C)$ if `side = 'L'`, or $\text{sub}(C) * Q^H$ if `side = 'R'`.

p?larzt

Forms the triangular factor T of a block reflector $H=I-V*T*V^H$ as returned by `p?tzrzf`.

Syntax

Fortran:

```
call pslarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pdlarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pclarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
call pzlarzt(direct, storev, n, k, v, iv, jv, descv, tau, t, work)
```

C:

```
void pslarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , float *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , float *tau , float *t , float *work );
void pdlarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , double *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , double *tau , double *t , double *work );
void pclarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , MKL_Complex8 *v ,
MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_Complex8 *tau , MKL_Complex8 *t ,
MKL_Complex8 *work );
void pzlarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , MKL_Complex16
*v , MKL_INT *iv , MKL_INT *jv , MKL_INT *descv , MKL_Complex16 *tau , MKL_Complex16
*t , MKL_Complex16 *work );
```

Include Files

- C: `mkl_scalapack.h`

Description

The `p?larzt` routine forms the triangular factor T of a real/complex block reflector H of order greater than n , which is defined as a product of k elementary reflectors as returned by `p?tzrzf`.

If `direct = 'F'`, $H = H(1)*H(2)*...*H(k)$, and T is upper triangular;

If `direct = 'B'`, $H = H(k)*...*H(2)*H(1)$, and T is lower triangular.

If `storev = 'C'`, the vector which defines the elementary reflector $H(i)$, is stored in the i -th column of the array `v`, and

$H = i - v * t * v'$.

If $storev = 'R'$, the vector, which defines the elementary reflector $H(i)$, is stored in the i -th row of the array v , and

$$H = i - v^* t^* v$$

Currently, only $storev = 'R'$ and $direct = 'B'$ are supported.

Input Parameters

<i>direct</i>	(global) CHARACTER.
	Specifies the order in which the elementary reflectors are multiplied to form the block reflector:
	if $direct = 'F'$: $H = H(1)*H(2)*\dots*H(k)$ (Forward, not supported)
	if $direct = 'B'$: $H = H(k)*\dots*H(2)*H(1)$ (Backward).
<i>storev</i>	(global) CHARACTER.
	Specifies how the vectors which defines the elementary reflectors are stored:
	if $storev = 'C'$: columnwise (not supported);
	if $storev = 'R'$: rowwise.
<i>n</i>	(global) INTEGER.
	The order of the block reflector H . $n \geq 0$.
<i>k</i>	(global) INTEGER.
	The order of the triangular factor T (= the number of elementary reflectors).
	$1 \leq k \leq mb_v (= nb_v)$.
<i>v</i>	REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt COMPLEX*16 for pzclarzt.
	Pointer into the local memory to an array of local size ($LOCr(iv+k-1)$, $LOCc(jv+n-1)$).
	The distributed matrix V contains the Householder vectors. See <i>Application Notes</i> below.
<i>iv</i> , <i>jv</i>	(global) INTEGER. The row and column indices in the global array V indicating the first row and the first column of the submatrix $\text{sub}(V)$, respectively.
<i>descv</i>	(local) INTEGER array, size (<i>dlen_</i>). The array descriptor for the distributed matrix V .
<i>tau</i>	(local) REAL for pslarzt DOUBLE PRECISION for pdlarzt COMPLEX for pclarzt

`COMPLEX*16` for `pzlarzt`.

Array, DIMENSION $LOCr(iv+k-1)$ if $incv = m_v$, and $LOCc(jv+k-1)$ otherwise. This array contains the Householder scalars related to the Householder vectors.

τ is tied to the distributed matrix V .

$work$

(local).

`REAL` for `pslarzt`

`DOUBLE PRECISION` for `pdlarzt`

`COMPLEX` for `pclarzt`

`COMPLEX*16` for `pzlarzt`.

Workspace array, size $(k*(k-1)/2)$.

Output Parameters

v

`REAL` for `pslarzt`

`DOUBLE PRECISION` for `pdlarzt`

`COMPLEX` for `pclarzt`

`COMPLEX*16` for `pzlarzt`.

t

(local)

`REAL` for `pslarzt`

`DOUBLE PRECISION` for `pdlarzt`

`COMPLEX` for `pclarzt`

`COMPLEX*16` for `pzlarzt`.

Array, size mb_v by mb_v . It contains the k -by- k triangular factor of the block reflector associated with v . t is lower triangular.

Application Notes

The shape of the matrix V and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

```
direct='F' and storev='C'

$$\begin{bmatrix} v1 & v2 & v3 \\ v1 & v2 & v3 \end{bmatrix}$$

v =   . . .
      . . .
      1 . .
          1 .
              1
```

direct='F' and storev='R':

$$\begin{bmatrix} & & & \overbrace{v_1 & v_1 & v_1 & v_1}^x & v_1 & \dots & 1 \\ v_1 & v_1 & v_1 & v_1 & v_1 & \dots & 1 \\ v_2 & v_2 & v_2 & v_2 & v_2 & \dots & 1 \\ v_3 & v_3 & v_3 & v_3 & v_3 & \dots & 1 \end{bmatrix}$$

direct='B' and storev='C':

$$\begin{array}{c} 1 \\ . \quad 1 \\ . \quad . \quad 1 \\ . \quad . \quad . \\ v = . \quad . \quad . \\ \begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \end{bmatrix} \end{array}$$

direct='B' and storev='R':

$$\begin{bmatrix} 1 & \dots & \overbrace{v_1 & v_1 & v_1 & v_1}^x & v_1 \\ . \quad 1 & \dots & v_2 & v_2 & v_2 & v_2 \\ . \quad . \quad 1 & \dots & v_3 & v_3 & v_3 & v_3 \end{bmatrix}$$

p?lascl

Multiplies a general rectangular matrix by a real scalar defined as C_{to}/C_{from} .

Syntax

Fortran:

```
call pslascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pdlascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pclascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
call pzlascl(type, cfrom, cto, m, n, a, ia, ja, desca, info)
```

C:

```
void pslascl (char *type , float *cfrom , float *cto , MKL_INT *m , MKL_INT *n , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
void pdlascl (char *type , double *cfrom , double *cto , MKL_INT *m , MKL_INT *n ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
void pclascl (char *type , float *cfrom , float *cto , MKL_INT *m , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
void pzlascl (char *type , double *cfrom , double *cto , MKL_INT *m , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?lascl` routine multiplies the m -by- n real/complex distributed matrix $\text{sub}(A)$ denoting $A(ia:ia+m-1, ja:ja+n-1)$ by the real/complex scalar $cto/cfrom$. This is done without over/underflow as long as the final result $cto \cdot A(i, j) / cfrom$ does not over/underflow. `type` specifies that $\text{sub}(A)$ may be full, upper triangular, lower triangular or upper Hessenberg.

Input Parameters

<code>type</code>	(global) CHARACTER. <code>type</code> indicates the storage type of the input distributed matrix. if <code>type</code> = 'G': $\text{sub}(A)$ is a full matrix, if <code>type</code> = 'L': $\text{sub}(A)$ is a lower triangular matrix, if <code>type</code> = 'U': $\text{sub}(A)$ is an upper triangular matrix, if <code>type</code> = 'H': $\text{sub}(A)$ is an upper Hessenberg matrix.
<code>cfrom, cto</code>	(global) REAL for <code>pslascl/pclascl</code> DOUBLE PRECISION for <code>pdlascl/pzlascl</code> . The distributed matrix $\text{sub}(A)$ is multiplied by $cto/cfrom$. $A(i,j)$ is computed without over/underflow if the final result $cto \cdot A(i,j) / cfrom$ can be represented without over/underflow. <code>cfrom</code> must be nonzero.
<code>m</code>	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).
<code>n</code>	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).
<code>a</code>	(local input/local output) REAL for <code>pslascl</code> DOUBLE PRECISION for <code>pdlascl</code> COMPLEX for <code>pclascl</code> COMPLEX*16 for <code>pzlascl</code> . Pointer into the local memory to an array of size <code>(lld_a, LOCc(ja+n-1))</code> . This array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<code>ia, ja</code>	(global) INTEGER. The column and row indices in the global array A indicating the first row and column of the submatrix $\text{sub}(A)$, respectively.
<code>desca</code>	(global and local) INTEGER.

Array of size (*dlen_*). The array descriptor for the distributed matrix *A*.

Output Parameters

- a* (local).
On exit, this array contains the local pieces of the distributed matrix multiplied by *cto/cfrom*.
- info* (local)
INTEGER.
if *info* = 0: the execution is successful.
if *info* < 0: If the *i*-th argument is an array and the *j*-entry had an illegal value, then *info* = -(*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value, then *info* = -*i*.

p?laset

Initializes the offdiagonal elements of a matrix to alpha and the diagonal elements to beta.

Syntax

Fortran:

```
call pslaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pdlaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pclaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
call pzlaset(uplo, m, n, alpha, beta, a, ia, ja, desca)
```

C:

```
void pslaset (char *uplo , MKL_INT *m , MKL_INT *n , float *alpha , float *beta ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pdlaset (char *uplo , MKL_INT *m , MKL_INT *n , double *alpha , double *beta ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pclaset (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex8 *alpha , MKL_Complex8
*beta , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pzlaset (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex16 *alpha ,
MKL_Complex16 *beta , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
```

Include Files

- C: mkl_scalapack.h

Description

The p?laset routine initializes an *m*-by-*n* distributed matrix sub(*A*) denoting *A*(*ia*:*ia+m-1*, *ja*:*ja+n-1*) to *beta* on the diagonal and *alpha* on the offdiagonals.

Input Parameters

- uplo* (global) CHARACTER.

Specifies the part of the distributed matrix $\text{sub}(A)$ to be set:

if $\text{uplo} = \text{'U'}$: upper triangular part; the strictly lower triangular part of $\text{sub}(A)$ is not changed;

if $\text{uplo} = \text{'L'}$: lower triangular part; the strictly upper triangular part of $\text{sub}(A)$ is not changed.

Otherwise: all of the matrix $\text{sub}(A)$ is set.

m

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).

n

(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).

alpha

(global).

REAL for pslaset

DOUBLE PRECISION for pdlaset

COMPLEX for pclaset

COMPLEX*16 for pzaset.

The constant to which the offdiagonal elements are to be set.

beta

(global).

REAL for pslaset

DOUBLE PRECISION for pdlaset

COMPLEX for pclaset

COMPLEX*16 for pzaset.

The constant to which the diagonal elements are to be set.

Output Parameters

a

(local).

REAL for pslaset

DOUBLE PRECISION for pdlaset

COMPLEX for pclaset

COMPLEX*16 for pzaset.

Pointer into the local memory to an array of size $(\text{lld}_a, \text{LOCc}(ja+n-1))$.

This array contains the local pieces of the distributed matrix $\text{sub}(A)$ to be set. On exit, the leading m -by- n submatrix $\text{sub}(A)$ is set as follows:

if $\text{uplo} = \text{'U'}$, $A(ia+i-1, ja+j-1) = \text{alpha}$, $1 \leq i \leq j-1$, $1 \leq j \leq n$,

if $\text{uplo} = \text{'L'}$, $A(ia+i-1, ja+j-1) = \text{alpha}$, $j+1 \leq i \leq m$, $1 \leq j \leq n$,

otherwise, $A(ia+i-1, ja+j-1) = \text{alpha}$, $1 \leq i \leq m$, $1 \leq j \leq n$, $ia+i.\text{ne}.ja+j$, and, for all uplo , $A(ia+i-1, ja+i-1) = \text{beta}$, $1 \leq i \leq \min(m, n)$.

<i>ia</i> , <i>ja</i>	(global) INTEGER.
	The column and row indices in the global array <i>A</i> indicating the first row and column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER.
	Array of size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .

p?lasmsub

Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.

Syntax

Fortran:

```
call pslasmsub(a, desca, i, l, k, smlnum, buf, lwork)
call pdlasmsub(a, desca, i, l, k, smlnum, buf, lwork)
```

C:

```
void pslasmsub (float *a , MKL_INT *desca , MKL_INT *i , MKL_INT *l , MKL_INT *k ,
float *smlnum , float *buf , MKL_INT *lwork );
void pdlasmsub (double *a , MKL_INT *desca , MKL_INT *i , MKL_INT *l , MKL_INT *k ,
double *smlnum , double *buf , MKL_INT *lwork );
```

Include Files

- C: mkl_scalapack.h

Description

The p?lasmsub routine looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero. This routine does a global maximum and must be called by all processes.

Input Parameters

<i>a</i>	(global) REAL for pslasmsub DOUBLE PRECISION for pdlasmsub Array, size (<i>desca(lld_)</i> , *).
	On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.
<i>desca</i>	(global and local) INTEGER. Array of size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .
<i>i</i>	(global) INTEGER. The global location of the bottom of the unreduced submatrix of <i>A</i> . Unchanged on exit.
<i>l</i>	(global) INTEGER. The global location of the top of the unreduced submatrix of <i>A</i> .

Unchanged on exit.

smlnum

(global)

REAL for `pslasmsub`

DOUBLE PRECISION for `pdlasmsub`

On entry, a "small number" for the given matrix. Unchanged on exit. A suggested value for *smlnum* is `slamch('s') * (n/slamch('p'))` for `pslasmsub` or `dlamch('s') * (n/dlamch('p'))` for `pdlasmsub`. See `lamch`.

lwork

(global) INTEGER.

On exit, *lwork* is the size of the work buffer.

This must be at least $2 \cdot \text{ceil}(\text{ceil}((i-1)/hbl) / \text{lcm}(nprow, npcol))$. Here `lcm` is least common multiple, and `nprow x npcol` is the logical grid size.

Output Parameters

k

(global) INTEGER.

On exit, this yields the bottom portion of the unreduced submatrix. This will satisfy: $1 \leq m \leq i-1$.

buf

(local).

REAL for `pslasmsub`

DOUBLE PRECISION for `pdlasmsub`

Array of size *lwork*.

p?lassq

Updates a sum of squares represented in scaled form.

Syntax

Fortran:

```
call pslassq(n, x, ix, jx, descx, incx, scale, sumsq)
call pdlassq(n, x, ix, jx, descx, incx, scale, sumsq)
call pclassq(n, x, ix, jx, descx, incx, scale, sumsq)
call pzlassq(n, x, ix, jx, descx, incx, scale, sumsq)
```

C:

```
void pslassq (MKL_INT *n , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , float *scale , float *sumsq );
void pdlassq (MKL_INT *n , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , double *scale , double *sumsq );
void pclassq (MKL_INT *n , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , float *scale , float *sumsq );
```

```
void pzlassq (MKL_INT *n , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , double *scale , double *sumsq );
```

Include Files

- C: mkl_scalapack.h

Description

The p?lassq routine returns the values *scl* and *sumsq* such that

$$scl^2 * sumsq = x(1)^2 + \dots + x(n)^2 + scale^2 * sumsq,$$

where $x(i) = \text{sub}(X) = X(ix + (jx-1)*descx(m_)) + (i - 1)*incx)$ for pslassq/pdlassq ,

and $x(i) = \text{abs}(X(ix + (jx-1)*descx(m_)) + (i - 1)*incx)$ for pclassq/pzlassq.

For real routines pslassq/pdlassq the value of *sumsq* is assumed to be non-negative and *scl* returns the value

$$scl = \max(scale, \text{abs}(x(i))).$$

For complex routines pclassq/pzlassq the value of *sumsq* is assumed to be at least unity and the value of *ssq* will then satisfy

$$1.0 \leq ssq \leq sumsq + 2n$$

Value *scale* is assumed to be non-negative and *scl* returns the value

$$scl = \max_i (scale, \text{abs}(\text{real}(x(i))), \text{abs}(\text{imag}(x(i))))$$

For all routines p?lassq values *scale* and *sumsq* must be supplied in *scale* and *sumsq* respectively, and *scale* and *sumsq* are overwritten by *scl* and *ssq* respectively.

All routines p?lassq make only one pass through the vector sub(x).

Input Parameters

n (global) INTEGER.

The length of the distributed vector sub(*x*).

x REAL for pslassq

DOUBLE PRECISION for pdlassq

COMPLEX for pclassq

COMPLEX*16 for pzlassq.

The vector for which a scaled sum of squares is computed:

$x(ix + (jx-1)*m_x + (i - 1)*incx), 1 \leq i \leq n.$

ix (global) INTEGER.

The row index in the global array *X* indicating the first row of sub(*X*).

jx (global) INTEGER.

The column index in the global array *X* indicating the first column of sub(*X*).

descx (global and local) INTEGER array of size (*dlen*_).

The array descriptor for the distributed matrix X .

incx (global) INTEGER.

The global increment for the elements of X . Only two values of *incx* are supported in this version, namely 1 and m_x . The argument *incx* must not equal zero.

scale (local).

REAL for pslassq/pclassq

DOUBLE PRECISION for pdlassq/pzlassq.

On entry, the value *scale* in the equation above.

sumsq (local)

REAL for pslassq/pclassq

DOUBLE PRECISION for pdlassq/pzlassq.

On entry, the value *sumsq* in the equation above.

Output Parameters

scale (local).

On exit, *scale* is overwritten with *scl*, the scaling factor for the sum of squares.

sumsq (local).

On exit, *sumsq* is overwritten with the value *smsq*, the basic sum of squares from which *scl* has been factored out.

p?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

Fortran:

```
call pslaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pdlaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pclaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
call pzlaswp(direc, rowcol, n, a, ia, ja, desca, k1, k2, ipiv)
```

C:

```
void pslaswp (char *direc , char *rowcol , MKL_INT *n , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *k1 , MKL_INT *k2 , MKL_INT *ipiv );
void pdlaswp (char *direc , char *rowcol , MKL_INT *n , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *k1 , MKL_INT *k2 , MKL_INT *ipiv );
void pclaswp (char *direc , char *rowcol , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *k1 , MKL_INT *k2 , MKL_INT *ipiv );
```

```
void pzlaswp (char *direc , char *rowcol , MKL_INT *n , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *k1 , MKL_INT *k2 , MKL_INT *ipiv );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?laswp` routine performs a series of row or column interchanges on the distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$. One interchange is initiated for each of rows or columns $k1$ through $k2$ of $\text{sub}(A)$. This routine assumes that the pivoting information has already been broadcast along the process row or column. Also note that this routine will only work for $k1-k2$ being in the same mb (or nb) block. If you want to pivot a full matrix, use `p?lapiv`.

Input Parameters

<code>direc</code>	(global) CHARACTER.
	Specifies in which order the permutation is applied:
	= 'F' - forward,
	= 'B' - backward.
<code>rowcol</code>	(global) CHARACTER.
	Specifies if the rows or columns are permuted:
	= 'R' - rows,
	= 'C' - columns.
<code>n</code>	(global) INTEGER.
	If <code>rowcol='R'</code> , the length of the rows of the distributed matrix $A(*, ja:ja+n-1)$ to be permuted;
	If <code>rowcol='C'</code> , the length of the columns of the distributed matrix $A(ia:ia+n-1, *)$ to be permuted;
<code>a</code>	(local)
	REAL for <code>pslaswp</code>
	DOUBLE PRECISION for <code>pdlaswp</code>
	COMPLEX for <code>pclaspw</code>
	COMPLEX*16 for <code>pzlaswp</code> .
	Pointer into the local memory to an array of size ($lld_a, *$). On entry, this array contains the local pieces of the distributed matrix to which the row/columns interchanges will be applied.
<code>ia</code>	(global) INTEGER.
	The row index in the global array A indicating the first row of $\text{sub}(A)$.
<code>ja</code>	(global) INTEGER.
	The column index in the global array A indicating the first column of $\text{sub}(A)$.
<code>desca</code>	(global and local) INTEGER array of size ($dlen_$).

The array descriptor for the distributed matrix A .

$k1$ (global) INTEGER.

The first element of $ipiv$ for which a row or column interchange will be done.

$k2$ (global) INTEGER.

The last element of $ipiv$ for which a row or column interchange will be done.

$ipiv$ (local)

INTEGER. Array, size $LOCr(m_a) + mb_a$ for row pivoting and $LOCr(n_a) + nb_a$ for column pivoting. This array is tied to the matrix A , $ipiv(k)=l$ implies rows (or columns) k and l are to be interchanged.

Output Parameters

A (local)

REAL for pslaswp

DOUBLE PRECISION for pdlaswp

COMPLEX for pcclaswp

COMPLEX*16 for pzblaswp.

On exit, the permuted distributed matrix.

p?latra

Computes the trace of a general square distributed matrix.

Syntax

Fortran:

```
val = pslatra(n, a, ia, ja, desca)
val = pdlatra(n, a, ia, ja, desca)
val = pclatra(n, a, ia, ja, desca)
val = pzlatra(n, a, ia, ja, desca)
```

C:

```
float pslatra (MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
double pdlatra (MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pclatra (MKL_Complex8 * , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca );
void pzlatra (MKL_Complex16 * , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca );
```

Include Files

- C: mkl_scalapack.h

Description

This function computes the trace of an n -by- n distributed matrix $\text{sub}(A)$ denoting $A(ia:ia+n-1, ja:ja+n-1)$. The result is left on every process of the grid.

Input Parameters

<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.
<i>a</i>	(local). REAL for pslatra DOUBLE PRECISION for pdlatra COMPLEX for pclatra COMPLEX*16 for pzlatra. Pointer into the local memory to an array of size (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>) containing the local pieces of the distributed matrix, the trace of which is to be computed.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices respectively in the global array <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .

Output Parameters

<i>val</i>	The value returned by the function.
------------	-------------------------------------

p?latrd

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

Syntax

Fortran:

```
call pslatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pdlatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pclatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
call pzlatrd(uplo, n, nb, a, ia, ja, desca, d, e, tau, w, iw, jw, descw, work)
```

C:

```
void pslatrd (char *uplo , MKL_INT *n , MKL_INT *nb , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *d , float *e , float *tau , float *w , MKL_INT *iw ,
MKL_INT *jw , MKL_INT *descw , float *work );
```

```

void pdlatrd (char *uplo , MKL_INT *n , MKL_INT *nb , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *d , double *e , double *tau , double *w ,
MKL_INT *iw , MKL_INT *jw , MKL_INT *descw , double *work );

void pclatrd (char *uplo , MKL_INT *n , MKL_INT *nb , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *d , float *e , MKL_Complex8 *tau , MKL_Complex8
*w , MKL_INT *iw , MKL_INT *jw , MKL_INT *descw , MKL_Complex8 *work );

void pzlatrd (char *uplo , MKL_INT *n , MKL_INT *nb , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *d , double *e , MKL_Complex16 *tau ,
MKL_Complex16 *w , MKL_INT *iw , MKL_INT *jw , MKL_INT *descw , MKL_Complex16 *work );

```

Include Files

- C: mkl_scalapack.h

Description

The p?latrd routine reduces nb rows and columns of a real symmetric or complex Hermitian matrix sub(A) = $A(ia:ia+n-1, ja:ja+n-1)$ to symmetric/complex tridiagonal form by an orthogonal/unitary similarity transformation $Q^* \text{sub}(A) Q$, and returns the matrices V and W , which are needed to apply the transformation to the unreduced part of sub(A).

If $uplo = U$, p?latrd reduces the last nb rows and columns of a matrix, of which the upper triangle is supplied;

if $uplo = L$, p?latrd reduces the first nb rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary routine called by p?syrtd/p?hetrd.

Input Parameters

uplo

(global) CHARACTER.

Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(A) is stored:

= 'U': Upper triangular
= 'L': Lower triangular.

n

(global) INTEGER.

The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub(A). $n \geq 0$.

nb

(global) INTEGER.

The number of rows and columns to be reduced.

a

REAL for pslatrd

DOUBLE PRECISION for pdlatrd

COMPLEX for pclatrd

COMPLEX*16 for pzlatrd.

Pointer into the local memory to an array of size(lld_a, LOCc(ja+n-1)).

On entry, this array contains the local pieces of the symmetric/Hermitian distributed matrix sub(A).

If $uplo = \text{U}$, the leading n -by- n upper triangular part of $\text{sub}(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If $uplo = \text{L}$, the leading n -by- n lower triangular part of $\text{sub}(A)$ contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia

(global) INTEGER.

The row index in the global array A indicating the first row of $\text{sub}(A)$.*ja*

(global) INTEGER.

The column index in the global array A indicating the first column of $\text{sub}(A)$.*desca*(global and local) INTEGER array of size ($dlen_$). The array descriptor for the distributed matrix A .*iw*

(global) INTEGER.

The row index in the global array W indicating the first row of $\text{sub}(W)$.*jw*

(global) INTEGER.

The column index in the global array W indicating the first column of $\text{sub}(W)$.*descw*(global and local) INTEGER array of size ($dlen_$). The array descriptor for the distributed matrix W .*work*

(local)

```
REAL for pslatrd  
DOUBLE PRECISION for pdlatrd  
COMPLEX for pclatrd  
COMPLEX*16 for pzlatrd.
```

Workspace array of size (nb_a).

Output Parameters

a

(local)

On exit, if $uplo = \text{'U'}$, the last nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of $\text{sub}(A)$; the elements above the diagonal with the array τau represent the orthogonal/unitary matrix Q as a product of elementary reflectors;

if $uplo = \text{'L'}$, the first nb columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of $\text{sub}(A)$; the elements below the diagonal with the array τau represent the orthogonal/unitary matrix Q as a product of elementary reflectors.

d

(local)

REAL for pslatrd/pclatrd

DOUBLE PRECISION for pdlatrd/pzlatrd.

Array, size $\text{LOCc}(ja+n-1)$.

The diagonal elements of the tridiagonal matrix T : $d(i) = a(i, i)$. d is tied to the distributed matrix A .

e

(local)

REAL for pslatrd/pclatrd

DOUBLE PRECISION for pdlatrd/pzlatrd.

Array, size $\text{LOCc}(ja+n-1)$ if $\text{uplo} = \text{'U'}$, $\text{LOCc}(ja+n-2)$ otherwise.

The off-diagonal elements of the tridiagonal matrix T :

$e(i) = a(i, i + 1)$ if $\text{uplo} = \text{'U'}$,

$e(i) = a(i + 1, i)$ if $\text{uplo} = \text{L}$.

e is tied to the distributed matrix A .

tau

(local)

REAL for pslatrd

DOUBLE PRECISION for pdlatrd

COMPLEX for pclatrd

COMPLEX*16 for pzlatrd.

Array, size $\text{LOCc}(ja+n-1)$. This array contains the scalar factors tau of the elementary reflectors. tau is tied to the distributed matrix A .

w

(local)

REAL for pslatrd

DOUBLE PRECISION for pdlatrd

COMPLEX for pclatrd

COMPLEX*16 for pzlatrd.

Pointer into the local memory to an array of size l/d_w by nb_w . This array contains the local pieces of the n -by- nb_w matrix w required to update the unreduced part of $\text{sub}(A)$.

Application Notes

If $\text{uplo} = \text{'U'}$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n) * H(n-1) * \dots * H(n-nb+1)$$

Each $H(i)$ has the form

$$H(i) = I - tau * v * v'$$

where tau is a real/complex scalar, and v is a real/complex vector with $v(i:n) = 0$ and $v(i-1) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-1, ja+i)$, and tau in $tau(ja+i-1)$.

If $\text{uplo} = \text{L}$, the matrix Q is represented as a product of elementary reflectors

$$Q = H(1) * H(2) * \dots * H(nb)$$

Each $H(i)$ has the form

$$H(i) = I - tau * v * v'$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1: ia+n-1, ja+i-1)$, and τ in $\tau(ia+i-1)$.

The elements of the vectors v together form the n -by- nb matrix V which is needed, with W , to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank- $2k$ update of the form:

$\text{sub}(A) := \text{sub}(A) - vw' - wv'$.

The contents of a on exit are illustrated by the following examples with

$n = 5$ and $nb = 2$:

$$\begin{array}{ll} \text{if } \text{uplo}=\text{U}: & \text{if } \text{uplo}=\text{L}: \\ \left[\begin{array}{ccccc} a & a & a & v_4 & v_5 \\ & a & a & v_4 & v_5 \\ & & a & 1 & v_5 \\ & & d & 1 & \\ & & & d & \end{array} \right] & \left[\begin{array}{ccccc} d & & & & \\ 1 & d & & & \\ v_1 & 1 & a & & \\ v_1 & v_2 & a & a & \\ v_1 & v_2 & a & a & a \end{array} \right] \end{array}$$

where d denotes a diagonal element of the reduced matrix, a denotes an element of the original matrix that is unchanged, and v_i denotes an element of the vector defining $H(i)$.

p?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

Syntax

Fortran:

```
call pslatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale,
cnorm, work)

call pdlatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale,
cnorm, work)

call pclatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale,
cnorm, work)

call pzlatrs(uplo, trans, diag, normin, n, a, ia, ja, desca, x, ix, jx, descx, scale,
cnorm, work)
```

C:

```
void pslatrs (char *uplo , char *trans , char *diag , char *normin , MKL_INT *n ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , float *scale , float *cnorm , float *work );

void pdlatrs (char *uplo , char *trans , char *diag , char *normin , MKL_INT *n ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , double *scale , double *cnorm , double *work );

void pclatrs (char *uplo , char *trans , char *diag , char *normin , MKL_INT *n ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *scale , float *cnorm ,
MKL_Complex8 *work );
```

```
void pzlatrs (char *uplo , char *trans , char *diag , char *normin , MKL_INT *n ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *x ,
MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *scale , double *cnorm ,
MKL_Complex16 *work );
```

Include Files

- C: mkl_scalapack.h

Description

The p?latrs routine solves a triangular system of equations $Ax = sb$, $A^T x = sb$ or $A^H x = sb$, where s is a scale factor set to prevent overflow. The description of the routine will be extended in the future releases.

Input Parameters

<i>uplo</i>	CHARACTER*1. Specifies whether the matrix A is upper or lower triangular. = 'U': Upper triangular = 'L': Lower triangular
<i>trans</i>	CHARACTER*1. Specifies the operation applied to Ax . = 'N': Solve $Ax = s*b$ (no transpose) = 'T': Solve $A^T x = s*b$ (transpose) = 'C': Solve $A^H x = s*b$ (conjugate transpose), where s - is a scale factor
<i>diag</i>	CHARACTER*1. Specifies whether or not the matrix A is unit triangular. = 'N': Non-unit triangular = 'U': Unit triangular
<i>normin</i>	CHARACTER*1. Specifies whether <i>cnorm</i> has been set or not. = 'Y': <i>cnorm</i> contains the column norms on entry; = 'N': <i>cnorm</i> is not set on entry. On exit, the norms will be computed and stored in <i>cnorm</i> .
<i>n</i>	INTEGER. The order of the matrix A . $n \geq 0$
<i>a</i>	REAL for pslatrs/pclatrs DOUBLE PRECISION for pdlatrs/pzlatrs Array, size <i>lda</i> by <i>n</i> . Contains the triangular matrix A . If <i>uplo</i> = U, the leading <i>n</i> -by- <i>n</i> upper triangular part of the array <i>a</i> contains the upper triangular matrix, and the strictly lower triangular part of <i>a</i> is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of the array a contains the lower triangular matrix, and the strictly upper triangular part of a is not referenced.

If $diag = 'U'$, the diagonal elements of a are also not referenced and are assumed to be 1.

ia, ja

(global) INTEGER. The row and column indices in the global array a indicating the first row and the first column of the submatrix A , respectively.

desca

(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix A .

x

REAL for pslatrs/pclatrs

DOUBLE PRECISION for pdlatrs/pzlatrs

Array, size (n). On entry, the right hand side b of the triangular system.

ix

(global) INTEGER. The row index in the global array x indicating the first row of $\text{sub}(x)$.

jx

(global) INTEGER.

The column index in the global array x indicating the first column of $\text{sub}(x)$.

descx

(global and local) INTEGER.

Array, size ($dlen_$). The array descriptor for the distributed matrix X .

cnorm

REAL for pslatrs/pclatrs

DOUBLE PRECISION for pdlatrs/pzlatrs.

Array, size (n). If $normin = 'Y'$, $cnorm$ is an input argument and $cnorm(j)$ contains the norm of the off-diagonal part of the j -th column of A . If $trans = 'N'$, $cnorm(j)$ must be greater than or equal to the infinity-norm, and if $trans = 'T'$ or ' C' , $cnorm(j)$ must be greater than or equal to the 1-norm.

work

(local).

REAL for pslatrs

DOUBLE PRECISION for pdlatrs

COMPLEX for pclatrs

COMPLEX*16 for pzlatrs.

Temporary workspace.

Output Parameters

X

On exit, x is overwritten by the solution vector x .

scale

REAL for pslatrs/pclatrs

DOUBLE PRECISION for pdlatrs/pzlatrs.

Array, size lda by n . The scaling factor s for the triangular system as described above.

If $scale = 0$, the matrix A is singular or badly scaled, and the vector x is an exact or approximate solution to $AX = 0$.

$cnorm$

If $normin = 'N'$, $cnorm$ is an output argument and $cnorm(j)$ returns the 1-norm of the off-diagonal part of the j -th column of A .

p?latrz

Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.

Syntax

Fortran:

```
call pslatrz(m, n, l, a, ia, ja, desca, tau, work)
call pdlatrz(m, n, l, a, ia, ja, desca, tau, work)
call pclatrz(m, n, l, a, ia, ja, desca, tau, work)
call pzlatrz(m, n, l, a, ia, ja, desca, tau, work)
```

C:

```
void pslatrz (MKL_INT *m , MKL_INT *n , MKL_INT *l , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work );
void pdlatrz (MKL_INT *m , MKL_INT *n , MKL_INT *l , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work );
void pclatrz (MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work );
void pzlatrz (MKL_INT *m , MKL_INT *n , MKL_INT *l , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?latrz` routine reduces the m -by- n ($m \leq n$) real/complex upper trapezoidal matrix $\text{sub}(A) = [A(ia:ia+m-1, ja:ja+m-1) A(ia:ia+m-1, ja+n-1:ja+n-1)]$ to upper triangular form by means of orthogonal/unitary transformations.

The upper trapezoidal matrix $\text{sub}(A)$ is factored as

$\text{sub}(A) = (R \ 0) * Z$,

where Z is an n -by- n orthogonal/unitary matrix and R is an m -by- m upper triangular matrix.

Input Parameters

m

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(A)$. $m \geq 0$.

n

(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.

l (global) INTEGER.

The number of columns of the distributed submatrix $\text{sub}(A)$ containing the meaningful part of the Householder reflectors. $l > 0$.

a (local)

REAL for pslatrz
DOUBLE PRECISION for pdlatrz
COMPLEX for pclatrz
COMPLEX*16 for pzlatrz.

Pointer into the local memory to an array of size (*lld_a*, *LOCc(ja+n-1)*). On entry, the local pieces of the m -by- n distributed matrix $\text{sub}(A)$, which is to be factored.

ia (global) INTEGER.

The row index in the global array *A* indicating the first row of $\text{sub}(A)$.

ja (global) INTEGER.

The column index in the global array *A* indicating the first column of $\text{sub}(A)$.

desca (global and local) INTEGER array of size (*dlen_*).

The array descriptor for the distributed matrix *A*.

work (local)

REAL for pslatrz
DOUBLE PRECISION for pdlatrz
COMPLEX for pclatrz
COMPLEX*16 for pzlatrz.

Workspace array, size (*lwork*).

lwork $\geq nq0 + \max(1, mp0)$, where
iroff = mod(*ia*-1, *mb_a*),
icoff = mod(*ja*-1, *nb_a*),
iarow = idxg2p(*ia*, *mb_a*, *myrow*, *rsrc_a*, *nproc*),
iacol = idxg2p(*ja*, *nb_a*, *mycol*, *csrc_a*, *ncol*),
mp0 = numroc(*m+iroff*, *mb_a*, *myrow*, *iarow*, *nproc*),
nq0 = numroc(*n+icoff*, *nb_a*, *mycol*, *iacol*, *ncol*),
numroc, *idxg2p*, and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nproc*, and *ncol* can be determined by calling the subroutine *blacs_gridinfo*.

Output Parameters

a

On exit, the leading m -by- m upper triangular part of $\text{sub}(A)$ contains the upper triangular matrix R , and elements $n-l+1$ to n of the first m rows of $\text{sub}(A)$, with the array τ_{au} , represent the orthogonal/unitary matrix Z as a product of m elementary reflectors.

tau

(local)

REAL for pslatrz

DOUBLE PRECISION for pdlatrz

COMPLEX for pclatrz

COMPLEX*16 for pzlatrz.

Array, size ($\text{LOC}_r(ja+m-1)$). This array contains the scalar factors of the elementary reflectors. τ_{au} is tied to the distributed matrix A .

Application Notes

The factorization is obtained by Householder's method. The k -th transformation matrix, $Z(k)$, which is used (or, in case of complex routines, whose conjugate transpose is used) to introduce zeros into the $(m - k + 1)$ -th row of $\text{sub}(A)$, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix},$$

where

$$T(k) = I - \tau_{\text{au}} * u(k) * u(k)', \quad u(k) = \begin{bmatrix} I \\ 0 \\ z(k) \end{bmatrix}$$

τ_{au} is a scalar and $z(k)$ is an $(n-m)$ -element vector. τ_{au} and $z(k)$ are chosen to annihilate the elements of the k -th row of $\text{sub}(A)$. The scalar τ_{au} is returned in the k -th element of τ_{au} and the vector $u(k)$ in the k -th row of $\text{sub}(A)$, such that the elements of $z(k)$ are in $a(k, m+1), \dots, a(k, n)$. The elements of R are returned in the upper triangular part of $\text{sub}(A)$.

Z is given by

$Z = Z(1)Z(2)\dots Z(m)$.

p?lauu2

Computes the product U^*U' or L'^*L , where U and L are upper or lower triangular matrices (local unblocked algorithm).

Syntax

Fortran:

```
call pslauu2(uplo, n, a, ia, ja, desca)
call pdlauu2(uplo, n, a, ia, ja, desca)
call pclauu2(uplo, n, a, ia, ja, desca)
```

```
call pzlaau2(uplo, n, a, ia, ja, desca)
```

C:

```
void pslauu2 (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pdlauu2 (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca );
void pclauu2 (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca );
void pzlaau2 (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca );
```

Include Files

- C: mkl_scalapack.h

Description

The p?lauu2 routine computes the product U^*U' or L^*L' , where the triangular factor U or L is stored in the upper or lower triangular part of the distributed matrix

$\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

If $uplo = 'U'$ or 'u', then the upper triangle of the result is stored, overwriting the factor U in $\text{sub}(A)$.

If $uplo = 'L'$ or '', then the lower triangle of the result is stored, overwriting the factor L in $\text{sub}(A)$.

This is the unblocked form of the algorithm, calling [BLAS Level 2 Routines](#). No communication is performed by this routine, the matrix to operate on should be strictly local to one process.

Input Parameters

uplo (global) CHARACTER*1.
Specifies whether the triangular factor stored in the *matrix* $\text{sub}(A)$ is upper or lower triangular:
= U: upper triangular
= L: lower triangular.

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the triangular factor U or L . $n \geq 0$.

a (local)
REAL for pslauu2
DOUBLE PRECISION for pdlauu2
COMPLEX for pclauu2
COMPLEX*16 for pzlaau2.
Pointer into the local memory to an array of size $(\text{lld}_a, \text{LOCc}(ja+n-1))$.
On entry, the local pieces of the triangular factor U or L .

ia (global) INTEGER.
The row index in the global array *A* indicating the first row of $\text{sub}(A)$.

<i>ja</i>	(global) INTEGER.
	The column index in the global array A indicating the first column of $\text{sub}(A)$.
<i>desca</i>	(global and local) INTEGER array of size (<i>dlen_</i>). The array descriptor for the distributed matrix A .

Output Parameters

<i>a</i>	(local)
	On exit, if <i>uplo</i> = 'U', the upper triangle of the distributed matrix $\text{sub}(A)$ is overwritten with the upper triangle of the product U^*U' ; if <i>uplo</i> = 'L', the lower triangle of $\text{sub}(A)$ is overwritten with the lower triangle of the product L'^*L .

p?lauum

Computes the product U^*U' or L'^*L , where U and L are upper or lower triangular matrices.

Syntax

Fortran:

```
call pslauum(uplo, n, a, ia, ja, desca)
call pdlauum(uplo, n, a, ia, ja, desca)
call pclauum(uplo, n, a, ia, ja, desca)
call pzlaum(uplo, n, a, ia, ja, desca)
```

C:

```
void pslauum (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca );
void pdlauum (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca );
void pclauum (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca );
void pzlaum (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca );
```

Include Files

- C: mkl_scalapack.h

Description

The p?lauum routine computes the product U^*U' or L'^*L , where the triangular factor U or L is stored in the upper or lower triangular part of the matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

If *uplo* = 'U' or 'u', then the upper triangle of the result is stored, overwriting the factor U in $\text{sub}(A)$. If *uplo* = 'L' or 'l', then the lower triangle of the result is stored, overwriting the factor L in $\text{sub}(A)$.

This is the blocked form of the algorithm, calling Level 3 PBLAS.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the triangular factor stored in the matrix sub(<i>A</i>) is upper or lower triangular: = 'U': upper triangular = 'L': lower triangular.
<i>n</i>	(global) INTEGER. The number of rows and columns to be operated on, that is, the order of the triangular factor <i>U</i> or <i>L</i> . $n \geq 0$.
<i>a</i>	(local) REAL for pslauum DOUBLE PRECISION for pdlauum COMPLEX for pclauum COMPLEX*16 for pzlaum. Pointer into the local memory to an array of size (<i>lld_a</i> , <i>LOCc(ja+n-1)</i>). On entry, the local pieces of the triangular factor <i>U</i> or <i>L</i> .
<i>ia</i>	(global) INTEGER. The row index in the global array <i>A</i> indicating the first row of sub(<i>A</i>).
<i>ja</i>	(global) INTEGER. The column index in the global array <i>A</i> indicating the first column of sub(<i>A</i>).
<i>desca</i>	(global and local) INTEGER array of size (<i>dlen_</i>). The array descriptor for the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	(local) On exit, if <i>uplo</i> = 'U', the upper triangle of the distributed matrix sub(<i>A</i>) is overwritten with the upper triangle of the product <i>U</i> * <i>U</i> ' ; if <i>uplo</i> = 'L', the lower triangle of sub(<i>A</i>) is overwritten with the lower triangle of the product <i>L</i> ' * <i>L</i> .
----------	--

p?lawil

Forms the Wilkinson transform.

Syntax

Fortran:

```
call pslawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
call pdlawil(ii, jj, m, a, desca, h44, h33, h43h34, v)
```

C:

```
void pslawil (MKL_INT *ii , MKL_INT *jj , MKL_INT *m , float *a , MKL_INT *desca ,
float *h44 , float *h33 , float *h43h34 , float *v );
void pdlawil (MKL_INT *ii , MKL_INT *jj , MKL_INT *m , double *a , MKL_INT *desca ,
double *h44 , double *h33 , double *h43h34 , double *v );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?lawil` routine gets the transform given by $h44$, $h33$, and $h43h34$ into v starting at row m .

Input Parameters

<i>ii</i>	(global) INTEGER. Row owner of $h(m+2, m+2)$.
<i>jj</i>	(global) INTEGER. Column owner of $h(m+2, m+2)$.
<i>m</i>	(global) INTEGER. On entry, the location from where the transform starts (row m). Unchanged on exit.
<i>a</i>	(global) REAL for <code>pslawil</code> DOUBLE PRECISION for <code>pdlawil</code> Array, size (<code>desca(lld_)</code> ,*). On entry, the Hessenberg matrix. Unchanged on exit.
<i>desca</i>	(global and local) INTEGER Array of size (<code>dlen_</code>). The array descriptor for the distributed matrix A . Unchanged on exit.
<i>h43h34</i>	(global) REAL for <code>pslawil</code> DOUBLE PRECISION for <code>pdlawil</code> These three values are for the double shift QR iteration. Unchanged on exit.

Output Parameters

<i>v</i>	(global) REAL for <code>pslawil</code> DOUBLE PRECISION for <code>pdlawil</code> Array of size 3 that contains the transform on output.
----------	--

p?org2l/p?ung2l

Generates all or part of the orthogonal/unitary matrix Q from a QL factorization determined by p?geqlf (unblocked algorithm).

Syntax

Fortran:

```
call psorg2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorg2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcung2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzung2l(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psorg2l (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdorg2l (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcung2l (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
void pzung2l (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?org2l/p?ung2l routine generates an m -by- n real/complex distributed matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors of order m :

$$Q = H(k) * \dots * H(2) * H(1) \text{ as returned by p?geqlf.}$$

Input Parameters

m

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix Q . $m \geq 0$.

n

(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix Q . $m \geq n \geq 0$.

k

(global) INTEGER.

The number of elementary reflectors whose product defines the matrix Q . $n \geq k \geq 0$.

a

REAL for psorg21
 DOUBLE PRECISION for pdorg21
 COMPLEX for pcung21
 COMPLEX*16 for pzung21.
 Pointer into the local memory to an array, size (*lld_a*, *LOCc(ja+n-1)*).
 On entry, the *j*-th column must contain the vector that defines the elementary reflector $H(j)$, $ja+n-k \leq j \leq ja+n-k$, as returned by [p?geqlf](#) in the *k* columns of its *distributed matrix* argument $A(ia:*, ja+n-k:ja+n-1)$.

ia

(global) INTEGER.
 The row index in the global array *A* indicating the first row of sub(*A*).

ja

(global) INTEGER.
 The column index in the global array *A* indicating the first column of sub(*A*).

desca

(global and local) INTEGER array of size (*dlen_*). The array descriptor for the distributed matrix *A*.

tau

(local)
 REAL for psorg21
 DOUBLE PRECISION for pdorg21
 COMPLEX for pcung21
 COMPLEX*16 for pzung21.
 Array, size *LOCc(ja+n-1)*.
 This array contains the scalar factor $\tau_{au}(j)$ of the elementary reflector $H(j)$, as returned by [p?geqlf](#).

work

(local)
 REAL for psorg21
 DOUBLE PRECISION for pdorg21
 COMPLEX for pcung21
 COMPLEX*16 for pzung21.
 Workspace array, size (*lwork*).

lwork

(local or global) INTEGER.
 The dimension of the array *work*.
lwork is local input and must be at least $lwork \geq mpa0 + \max(1, nqa0)$, where
 $iroffa = \text{mod}(ia-1, mb_a),$
 $icoffa = \text{mod}(ja-1, nb_a),$
 $iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow),$
 $iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol),$

```

mpa0 = numroc(m+iroffa, mb_a, myrow, iarow, nprow),
nqa0 = numroc(n+icoffa, nb_a, mycol, iacol, npcol).

indxg2p and numroc are ScaLAPACK tool functions; myrow, mycol, nprow,
and npcol can be determined by calling the subroutine blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is
assumed; the routine only calculates the minimum and optimal size for all
work arrays. Each of these values is returned in the first entry of the
corresponding work array, and no error message is issued by pxerbla.

```

Output Parameters

<i>a</i>	On exit, this array contains the local pieces of the m -by- n distributed matrix Q .
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> *100 + <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?org2r/p?ung2r

Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by p?geqr (unblocked algorithm).

Syntax

Fortran:

```

call psorg2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorg2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcung2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzung2r(m, n, k, a, ia, ja, desca, tau, work, lwork, info)

```

C:

```

void psorg2r (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdorg2r (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcung2r (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
void pzung2r (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The `p?org2r/p?ung2r` routine generates an m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m :

$$Q = H(1) * H(2) * \dots * H(k)$$

as returned by `p?geqrf`.

Input Parameters

`m`

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix Q . $m \geq 0$.

`n`

(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix Q . $m \geq n \geq 0$.

`k`

(global) INTEGER.

The number of elementary reflectors whose product defines the matrix Q . $n \geq k \geq 0$.

`a`

REAL for `psorg2r`

DOUBLE PRECISION for `pdorg2r`

COMPLEX for `pcung2r`

COMPLEX*16 for `pzung2r`.

Pointer into the local memory to an array, size (`lld_a`, `LOCc(ja+n-1)`).

On entry, the j -th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by `p?geqrf` in the k columns of its *distributed matrix* argument $A(ia:*, ja:ja+k-1)$.

`ia`

(global) INTEGER.

The row index in the global array A indicating the first row of $\text{sub}(A)$.

`ja`

(global) INTEGER.

The column index in the global array A indicating the first column of $\text{sub}(A)$.

`desca`

(global and local) INTEGER array of size (`dlen_`).

The array descriptor for the distributed matrix A .

`tau`

(local)

REAL for `psorg2r`

DOUBLE PRECISION for `pdorg2r`

COMPLEX for `pcung2r`

`COMPLEX*16` for `pzung2r`.

Array, size $LOCc(ja+k-1)$.

This array contains the scalar factor $\tau(j)$ of the elementary reflector $H(j)$, as returned by `p?geqrf`. This array is tied to the distributed matrix A .

`work`

(local)

`REAL` for `psorg2r`

`DOUBLE PRECISION` for `pdorg2r`

`COMPLEX` for `pcung2r`

`COMPLEX*16` for `pzung2r`.

Workspace array, size ($lwork$).

`lwork`

(local or global) `INTEGER`.

The dimension of the array `work`.

$lwork$ is local input and must be at least $lwork \geq mpa0 + \max(1, nqa0)$,

where

```
iroffa = mod(ia-1, mb_a , icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, myrow, rsrc_a, nprow),
iacol = indxg2p(ja, nb_a, mycol, csrc_a, npcol),
mpa0 = numroc(m+iroffa, mb_a, myrow, iarow, nprow),
nqa0 = numroc(n+icoffa, nb_a, mycol, iacol, npcol).
```

`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the subroutine `blacs_gridinfo`.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

Output Parameters

`a`

On exit, this array contains the local pieces of the m -by- n distributed matrix Q .

`work`

On exit, `work(1)` returns the minimal and optimal $lwork$.

`info`

(local) `INTEGER`.

= 0: successful exit

< 0: if the i -th argument is an array and the j -entry had an illegal value,
then `info = - (i*100+j)`,

if the i -th argument is a scalar and had an illegal value,
then `info = -i`.

p?orgl2/p?ungl2

Generates all or part of the orthogonal/unitary matrix Q from an LQ factorization determined by p?gelqf (unblocked algorithm).

Syntax

Fortran:

```
call psorgl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungl2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psorgl2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdorgl2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcungl2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
void pzungl2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?orgl2/p?ungl2 routine generates a m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first m rows of a product of k elementary reflectors of order n

$$Q = H(k) * \dots * H(2) * H(1) \text{ (for real flavors),}$$

$$Q = (H(k))^H * \dots * (H(2))^H * (H(1))^H \text{ (for complex flavors) as returned by p?gelqf.}$$

Input Parameters

m

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix Q . $m \geq 0$.

n

(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix Q . $n \geq m \geq 0$.

k

(global) INTEGER.

The number of elementary reflectors whose product defines the matrix Q . $m \geq k \geq 0$.

<i>a</i>	REAL for psorgl2 DOUBLE PRECISION for pdorgl2 COMPLEX for pcungl2 COMPLEX*16 for pzungl2. Pointer into the local memory to an array, size ($\text{lld}_a, \text{LOCc}(ja+n-1)$). On entry, the i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the k rows of its <i>distributed matrix</i> argument $A(ia:ia+k-1, ja:*)$.
<i>ia</i>	(global) INTEGER. The row index in the global array A indicating the first row of $\text{sub}(A)$.
<i>ja</i>	(global) INTEGER. The column index in the global array A indicating the first column of $\text{sub}(A)$.
<i>desca</i>	(global and local) INTEGER array of size ($dlen_$). The array descriptor for the distributed matrix A .
<i>tau</i>	(local) REAL for psorgl2 DOUBLE PRECISION for pdorgl2 COMPLEX for pcungl2 COMPLEX*16 for pzungl2. Array, size $\text{LOCr}(ja+k-1)$. This array contains the scalar factors τ_{ai} of the elementary reflectors $H(i)$, as returned by p?gelqf . This array is tied to the distributed matrix A .
<i>WORK</i>	(local) REAL for psorgl2 DOUBLE PRECISION for pdorgl2 COMPLEX for pcungl2 COMPLEX*16 for pzungl2. Workspace array, size (lwork).
<i>lwork</i>	(local or global) INTEGER. The dimension of the array $work$. lwork is local input and must be at least $\text{lwork} \geq nqa0 + \max(1, mpa0)$, where $\begin{aligned} iroffa &= \text{mod}(ia-1, mb_a), \\ icoffa &= \text{mod}(ja-1, nb_a), \\ iarow &= \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow), \end{aligned}$

```

iacol = indxg2p(ja, nb_a, mycol, csrc_a, npcol),
mpa0 = numroc(m+iroffa, mb_a, myrow, iarow, nprow),
nqa0 = numroc(n+icoffa, nb_a, mycol, iacol, npcol).
indxg2p and numroc are ScaLAPACK tool functions; myrow, mycol, nprow,
and npcol can be determined by calling the subroutine blacs_gridinfo.
```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>a</i>	On exit, this array contains the local pieces of the <i>m</i> -by- <i>n</i> distributed matrix <i>Q</i> .
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (i*100+j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

p?org2/p?ungr2

Generates all or part of the orthogonal/unitary matrix *Q* from an RQ factorization determined by [p?gerqf](#) (unblocked algorithm).

Syntax

Fortran:

```
call psorgr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pdorgr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pcungr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
call pzungr2(m, n, k, a, ia, ja, desca, tau, work, lwork, info)
```

C:

```
void psorgr2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdorgr2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
void pcungr2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );
```

```
void pzungr2 (MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?orgr2/p?ungr2 routine generates an m -by- n real/complex matrix Q denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the last m rows of a product of k elementary reflectors of order n

$Q = H(1)^H * H(2)^H * \dots * H(k)^H$ (for real flavors);

$Q = (H(1))^H * (H(2))^H * \dots * (H(k))^H$ (for complex flavors) as returned by [p?gerqf](#).

Input Parameters

m	(global) INTEGER.
	The number of rows to be operated on, that is, the number of rows of the distributed submatrix Q . $m \geq 0$.
n	(global) INTEGER.
	The number of columns to be operated on, that is, the number of columns of the distributed submatrix Q . $n \geq m \geq 0$.
k	(global) INTEGER.
	The number of elementary reflectors whose product defines the matrix Q . $m \geq k \geq 0$.
a	REAL for psorgr2 DOUBLE PRECISION for pdorgr2 COMPLEX for pcungqr2 COMPLEX*16 for pzungr2. Pointer into the local memory to an array, size (lld_a , $LOCc(ja+n-1)$). On entry, the i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia+m-k \leq i \leq ia+m-1$, as returned by p?gerqf in the k rows of its <i>distributed matrix</i> argument $A(ia+m-k:ia+m-1, ja:*)$.
ia	(global) INTEGER. The row index in the global array A indicating the first row of sub(A).
ja	(global) INTEGER. The column index in the global array A indicating the first column of sub(A).
$desca$	(global and local) INTEGER array of size ($dlen_$). The array descriptor for the distributed matrix A .
tau	(local) REAL for psorgr2

DOUBLE PRECISION for pdorgqr2

COMPLEX for pcungqr2

COMPLEX*16 for pzungr2.

Array, size $LOC_r(ja+m-1)$. This array contains the scalar factors $tau(i)$ of the elementary reflectors $H(i)$, as returned by [p?gerqf](#). This array is tied to the distributed matrix A .

work

(local)

REAL for psorgqr2

DOUBLE PRECISION for pdorgqr2

COMPLEX for pcungqr2

COMPLEX*16 for pzungr2.

Workspace array, size (*lwork*).

lwork

(local or global) INTEGER.

The dimension of the array *work*.

lwork is local input and must be at least $lwork \geq nqa0 + \max(1, mpa0)$, where $iroffa = \text{mod}(ia-1, mb_a)$, $icoffa = \text{mod}(ja-1, nb_a)$,

$iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow)$,

$iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol)$,

$mpa0 = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow)$,

$nqa0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcol)$.

`indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprow`, and `npcol` can be determined by calling the subroutine `blacs_gridinfo`.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

a

On exit, this array contains the local pieces of the m -by- n distributed matrix Q .

work

On exit, *work*(1) returns the minimal and optimal *lwork*.

info

(local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,

then *info* = - (i*100+j),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -i.

p?orm2l/p?unm2l

Multiples a general matrix by the orthogonal/unitary matrix from a QL factorization determined by p?geqlf (unblocked algorithm).

Syntax

Fortran:

```
call psorm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pdorm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pcunm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pzunm2l(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

C:

```
void psorm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );

void pdorm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );

void pcunm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?orm2l/p?unm2l routine overwrites the general real/complex m -by- n distributed matrix sub $(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

$Q^* \text{sub}(C)$ if $\text{side} = 'L'$ and $\text{trans} = 'N'$, or

$Q^T \text{sub}(C) / Q^H \text{sub}(C)$ if $\text{side} = 'L'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors), or

$\text{sub}(C) * Q$ if $\text{side} = 'R'$ and $\text{trans} = 'N'$, or

$\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$ if $\text{side} = 'R'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(k) * \dots * H(2) * H(1)$ as returned by [p?geqlf](#). Q is of order m if $\text{side} = 'L'$ and of order n if $\text{side} = 'R'$.

Input Parameters

side	(global) CHARACTER. = ' L ': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left, = ' R ': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.
trans	(global) CHARACTER. = ' N ': apply Q (no transpose) = ' T ': apply Q^T (transpose, for real flavors) = ' C ': apply Q^H (conjugate transpose, for complex flavors)
m	(global) INTEGER. The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. $m \geq 0$.
n	(global) INTEGER. The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. $n \geq 0$.
k	(global) INTEGER. The number of elementary reflectors whose product defines the matrix Q . If $\text{side} = 'L'$, $m \geq k \geq 0$; if $\text{side} = 'R'$, $n \geq k \geq 0$.
a	(local) REAL for psorm2l DOUBLE PRECISION for pdorm2l COMPLEX for pcunm2l COMPLEX*16 for pzunm2l . Pointer into the local memory to an array, $\text{size}(\text{lld}_a, \text{LOC}_c(ja+k-1))$. On entry, the j -th row must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by p?geqlf in the k columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. The argument $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit. If $\text{side} = 'L'$, $\text{lld}_a \geq \max(1, \text{LOC}_r(ia+m-1))$, if $\text{side} = 'R'$, $\text{lld}_a \geq \max(1, \text{LOC}_r(ia+n-1))$.
ia	(global) INTEGER. The row index in the global array A indicating the first row of $\text{sub}(A)$.
ja	(global) INTEGER.

The column index in the global array A indicating the first column of $\text{sub}(A)$.

descA (global and local) INTEGER array of size (*dlen_*). The array descriptor for the distributed matrix A .

tau (local)

```
REAL for psorm2l
DOUBLE PRECISION for pdorm2l
COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
```

Array, size $LOC_C(ja+n-1)$. This array contains the scalar factor $\tau_{au}(j)$ of the elementary reflector $H(j)$, as returned by [p?geqlf](#). This array is tied to the distributed matrix A .

c (local)

```
REAL for psorm2l
DOUBLE PRECISION for pdorm2l
COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
```

Pointer into the local memory to an array, size (*lld_c*, $LOC_C(jc+n-1)$). On entry, the local pieces of the distributed matrix $\text{sub}(C)$.

ic (global) INTEGER.

The row index in the global array C indicating the first row of $\text{sub}(C)$.

jc (global) INTEGER.

The column index in the global array C indicating the first column of $\text{sub}(C)$.

descC (global and local) INTEGER array of size (*dlen_*). The array descriptor for the distributed matrix C .

work (local)

```
REAL for psorm2l
DOUBLE PRECISION for pdorm2l
COMPLEX for pcunm2l
COMPLEX*16 for pzunm2l.
```

Workspace array, size (*lwork*).

On exit, *work*(1) returns the minimal and optimal *lwork*.

lwork (local or global) INTEGER.

The dimension of the array *work*.

lwork is local input and must be at least

```
if side = 'L', lwork ≥ mpc0 + max(1, nqc0),
if side = 'R', lwork ≥ nqc0 + max(max(1, mpc0), numroc(numroc(n
+icoffc, nb_a, 0, 0, ncol), nb_a, 0, 0, lcmq)),
```

where

```

lcmq = lcm/npcol,
lcm = iclm(nprow, npcol),
iroffc = mod(ic-1, mb_c),
icooffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow),
iccol = indxg2p(jc, nb_c, mycol, csrc_c, npcol),
Mqc0 = numroc(m+icooffc, nb_c, mycol, icrow, nprow),
Npc0 = numroc(n+iroffc, mb_c, myrow, iccol, npcol),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions; myrow, mycol,
nprow, and npcol can be determined by calling the subroutine
blacs_gridinfo.
```

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

<i>c</i>	On exit, <i>c</i> is overwritten by $Q^* \text{sub}(C)$, or $Q^T * \text{sub}(C) / Q^H * \text{sub}(C)$, or $\text{sub}(C) * Q$, or $\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (i*100+j), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

```

If side = 'L', ( mb_a.eq.mb_c .AND. iroffa.eq.iroffc .AND. iarow.eq.icrow )
If side = 'R', ( mb_a.eq.nb_c .AND. iroffa.eq.iroffc ).
```

p?orm2r/p?unm2r

Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by p?geqr (unblocked algorithm).

Syntax

Fortran:

```
call psorm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pdorm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pcunm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pzunm2r(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

C:

```
void psorm2r (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );

void pdorm2r (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );

void pcunm2r (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );

void pzunm2r (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?orm2r/p?unm2r routine overwrites the general real/complex m -by- n distributed matrix sub $(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

$Q^* \text{sub}(C)$ if $\text{side} = 'L'$ and $\text{trans} = 'N'$, or

$Q^T \text{sub}(C) / Q^H \text{sub}(C)$ if $\text{side} = 'L'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors), or

$\text{sub}(C) * Q$ if $\text{side} = 'R'$ and $\text{trans} = 'N'$, or

$\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$ if $\text{side} = 'R'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary matrix defined as the product of k elementary reflectors

$Q = H(k) * \dots * H(2) * H(1)$ as returned by p?geqrf . Q is of order m if $\text{side} = 'L'$ and of order n if $\text{side} = 'R'$.

Input Parameters

side (global) CHARACTER.

= 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left,
 = 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.

trans

(global) CHARACTER.
 = 'N': apply Q (no transpose)
 = 'T': apply Q^T (transpose, for real flavors)
 = 'C': apply Q^H (conjugate transpose, for complex flavors)

m

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. $m \geq 0$.

n

(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. $n \geq 0$.

k

(global) INTEGER.

The number of elementary reflectors whose product defines the matrix Q .

```
If side = 'L', m ≥ k ≥ 0;  
if side = 'R', n ≥ k ≥ 0.
```

a

(local)

```
REAL for psorm2r  
DOUBLE PRECISION for pdorm2r  
COMPLEX for pcunm2r  
COMPLEX*16 for pzunm2r.
```

Pointer into the local memory to an array, size(*lld_a*, *LOCc(ja+k-1)*).

On entry, the *j*-th column must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by [p?geqrf](#) in the *k* columns of its distributed matrix argument $A(ia:*, ja:ja+k-1)$. The argument $A(ia:*, ja:ja+k-1)$ is modified by the routine but restored on exit.

```
If side = 'L', lld_a ≥ max(1, LOCr(ia+m-1)),  
if side = 'R', lld_a ≥ max(1, LOCr(ia+n-1)).
```

ia

(global) INTEGER.

The row index in the global array A indicating the first row of $\text{sub}(A)$.

ja

(global) INTEGER.

The column index in the global array A indicating the first column of $\text{sub}(A)$.

desca

(global and local) INTEGER array of size (*dlen_*). The array descriptor for the distributed matrix A .

tau

(local)

REAL **for** psorm2r
 DOUBLE PRECISION **for** pdorm2r
 COMPLEX **for** pcunm2r
 COMPLEX*16 **for** pzunm2r.

Array, size $LOC_C(ja+k-1)$. This array contains the scalar factors $\tau_{au}(j)$ of the elementary reflector $H(j)$, as returned by [p?geqrf](#). This array is tied to the distributed matrix A .

c
 (local)
 REAL **for** psorm2r
 DOUBLE PRECISION **for** pdorm2r
 COMPLEX **for** pcunm2r
 COMPLEX*16 **for** pzunm2r.
 Pointer into the local memory to an array, size ($l1d_c$, $LOC_C(jc+n-1)$).
 On entry, the local pieces of the distributed matrix sub (C).

ic
 (global) INTEGER.
 The row index in the global array C indicating the first row of sub(C).

jc
 (global) INTEGER.
 The column index in the global array C indicating the first column of sub(C).

descC
 (global and local) INTEGER array of size ($dlen_$).
 The array descriptor for the distributed matrix C .

work
 (local)
 REAL **for** psorm2r
 DOUBLE PRECISION **for** pdorm2r
 COMPLEX **for** pcunm2r
 COMPLEX*16 **for** pzunm2r.
 Workspace array, size ($lwork$).

lwork
 (local or global) INTEGER.
 The dimension of the array *work*.
lwork is local input and must be at least

```
if side = 'L', lwork ≥ mpc0 + max(1, nqc0),
if side = 'R', lwork ≥ nqc0 + max(max(1, mpc0), numroc(numroc(n
+icoffc, nb_a, 0, 0, npcol), nb_a, 0, 0, lcmq)),
```

 where

$$lcmq = lcm/npcol,$$

$$lcm = iclm(nprow, npcol),$$

$$iroffc = \text{mod}(ic-1, mb_c),$$

```

 $icoffc = \text{mod}(jc-1, nb\_c),$ 
 $icrow = \text{indxg2p}(ic, mb\_c, myrow, rsrc\_c, nprow),$ 
 $iccol = \text{indxg2p}(jc, nb\_c, mycol, csrc\_c, npcol),$ 
 $Mqc0 = \text{numroc}(m+icoffc, nb\_c, mycol, icrow, nprow),$ 
 $Npc0 = \text{numroc}(n+iroffc, mb\_c, myrow, iccol, npcol),$ 
ilcm, indxg2p and numroc are ScaLAPACK tool functions; myrow, mycol, nprow, and npcol can be determined by calling the subroutine blacs_gridinfo.

```

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by [pxerbla](#).

Output Parameters

c On exit, *c* is overwritten by $Q^* \text{sub}(C)$, or $Q^T * \text{sub}(C) / Q^H * \text{sub}(C)$, or $\text{sub}(C) * Q$, or $\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$

work On exit, *work*(1) returns the minimal and optimal *lwork*.

info (local) INTEGER.
= 0: successful exit
< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = - (*i**100+*j*),
if the *i*-th argument is a scalar and had an illegal value,
then *info* = -*i*.

NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', $(mb_a.eq.mb_c .AND. iroffa.eq.iroffc .AND. iarow.eq.icrow)$
If *side* = 'R', $(mb_a.eq.nb_c .AND. iroffa.eq.iroffc)$.

p?orml2/p?unml2

Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by p?gelqf (unblocked algorithm).

Syntax

Fortran:

```

call psorml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pdorml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

```

```
call pcunml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

```
call pzunml2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

C:

```
void psorml2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );
void pdorml2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );
void pcunml2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );
void pzunml2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?orml2/p?unml2 routine overwrites the general real/complex m -by- n distributed matrix sub $(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

$Q^* \text{sub}(C)$ if $\text{side} = 'L'$ and $\text{trans} = 'N'$, or

$Q^T \text{sub}(C) / Q^H \text{sub}(C)$ if $\text{side} = 'L'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors), or

$\text{sub}(C)^*Q$ if $\text{side} = 'R'$ and $\text{trans} = 'N'$, or

$\text{sub}(C)^*Q^T / \text{sub}(C)^*Q^H$ if $\text{side} = 'R'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(k)^* \dots ^* H(2)^* H(1)$ (for real flavors)

$Q = (H(k))^H \star \dots \star (H(2))^H \star (H(1))^H$ (for complex flavors)

as returned by p?gelqf . Q is of order m if $\text{side} = 'L'$ and of order n if $\text{side} = 'R'$.

Input Parameters

<i>side</i>	(global) CHARACTER.
	= 'L': apply Q or Q^T for real flavors (Q^H for complex flavors) from the left,
	= 'R': apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.
<i>trans</i>	(global) CHARACTER.

= 'N': apply Q (no transpose)
 = 'T': apply Q^T (transpose, for real flavors)
 = 'C': apply Q^H (conjugate transpose, for complex flavors)

m

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. $m \geq 0$.

n

(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. $n \geq 0$.

k

(global) INTEGER.

The number of elementary reflectors whose product defines the matrix Q .

If $\text{side} = 'L'$, $m \geq k \geq 0$;

if $\text{side} = 'R'$, $n \geq k \geq 0$.

a

(local)

REAL for psorml2

DOUBLE PRECISION for pdorml2

COMPLEX for pcunml2

COMPLEX*16 for pzunml2.

Pointer into the local memory to an array, size

(*lld_a*, $\text{LOCc}(ja+m-1)$ if $\text{side}='L'$,

(*lld_a*, $\text{LOCc}(ja+n-1)$ if $\text{side}='R'$,

where $\text{lld}_a \geq \max(1, \text{LOCr}(ia+k-1))$.

On entry, the i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by [p?gelqf](#) in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. The argument $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.

ia

(global) INTEGER.

The row index in the global array A indicating the first row of $\text{sub}(A)$.

ja

(global) INTEGER.

The column index in the global array A indicating the first column of $\text{sub}(A)$.

desca

(global and local) INTEGER array of size (*dlen_*). The array descriptor for the distributed matrix A .

tau

(local)

REAL for psorml2

DOUBLE PRECISION for pdorml2

COMPLEX for pcunml2

COMPLEX*16 **for** pzunml2.
 Array, size $\text{LOCc}(ia+k-1)$. This array contains the scalar factors $\tau(i)$ of the elementary reflector $H(i)$, as returned by [p?gelqf](#). This array is tied to the distributed matrix A .

c
 (local)
 REAL **for** psorml2
 DOUBLE PRECISION **for** pdorml2
 COMPLEX **for** pcunml2
 COMPLEX*16 **for** pzunml2.
 Pointer into the local memory to an array, size (lld_c , $\text{LOCc}(jc+n-1)$). On entry, the local pieces of the distributed matrix sub (C).

ic
 (global) INTEGER.
 The row index in the global array C indicating the first row of sub(C).

jc
 (global) INTEGER.
 The column index in the global array C indicating the first column of sub(C).

descC
 (global and local) INTEGER array of size ($dlen_$). The array descriptor for the distributed matrix C .

work
 (local)
 REAL **for** psorml2
 DOUBLE PRECISION **for** pdorml2
 COMPLEX **for** pcunml2
 COMPLEX*16 **for** pzunml2.
 Workspace array, size ($lwork$).

lwork
 (local or global) INTEGER.
 The dimension of the array *work*.
lwork is local input and must be at least

$$\text{if } side = 'L', lwork \geq mqc0 + \max(\max(1, nqc0), \text{numroc}(\text{numroc}(m + ioffc, mb_a, 0, 0, nprow), mb_a, 0, 0, lcm)) ,$$

$$\text{if } side = 'R', lwork \geq nqc0 + \max(1, mqc0) ,$$
 where

$$lcm = lcm / nprow,$$

$$lcm = iclm(nprow, nqcol),$$

$$iroffc = \text{mod}(ic-1, mb_c),$$

$$icoffc = \text{mod}(jc-1, nb_c),$$

$$icrow = \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprow),$$

$$iccol = \text{indxg2p}(jc, nb_c, mycol, csrc_c, nqcol),$$

$$Mpc0 = \text{numroc}(m + icoffc, mb_c, mycol, icrow, nprow),$$

$Nqc0 = \text{numroc}(n+iroffc, nb_c, myrow, iccol, npcol),$
ilcm, *indxg2p* and *numroc* are ScaLAPACK tool functions; *myrow*, *mycol*, *nprov*, and *npcol* can be determined by calling the subroutine *blacs_gridinfo*.

If *lwork* = -1, then *lwork* is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

Output Parameters

<i>c</i>	On exit, <i>c</i> is overwritten by $Q^* \text{sub}(C)$, or $Q^T * \text{sub}(C) / Q^H * \text{sub}(C)$, or $\text{sub}(C) * Q$, or $\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$
<i>work</i>	On exit, <i>work</i> (1) returns the minimal and optimal <i>lwork</i> .
<i>info</i>	(local) INTEGER. = 0: successful exit < 0: if the <i>i</i> -th argument is an array and the <i>j</i> -entry had an illegal value, then <i>info</i> = - (<i>i</i> *100+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> .

NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

If *side* = 'L', $(nb_a.eq.mb_c .AND. icoffa.eq.iroffc)$
If *side* = 'R', $(nb_a.eq.nb_c .AND. icoffa.eq.icoffc .AND. iacol.eq.iccol)$.

p?ormr2/p?unmr2

Multiples a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by p?gerqf (unblocked algorithm).

Syntax

Fortran:

```
call psormr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pdormr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pcunmr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)

call pzunmr2(side, trans, m, n, k, a, ia, ja, desca, tau, c, ic, jc, descc, work,
lwork, info)
```

C:

```

void psormr2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *desc , float *work , MKL_INT *lwork , MKL_INT *info );
void pdormr2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *desc , double *work , MKL_INT *lwork , MKL_INT *info );
void pcunmr2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );
void pzunmr2 (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau ,
MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *desc , MKL_Complex16 *work ,
MKL_INT *lwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The p?ormr2/p?unmr2 routine overwrites the general real/complex m -by- n distributed matrix sub $(C) = C(ic:ic+m-1, jc:jc+n-1)$ with

$Q^* \text{sub}(C)$ if $\text{side} = 'L'$ and $\text{trans} = 'N'$, or

$Q^T \text{sub}(C) / Q^H \text{sub}(C)$ if $\text{side} = 'L'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors), or

$\text{sub}(C) * Q$ if $\text{side} = 'R'$ and $\text{trans} = 'N'$, or

$\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$ if $\text{side} = 'R'$ and $\text{trans} = 'T'$ (for real flavors) or $\text{trans} = 'C'$ (for complex flavors).

where Q is a real orthogonal or complex unitary distributed matrix defined as the product of k elementary reflectors

$Q = H(1) * H(2) * \dots * H(k)$ (for real flavors)

$Q = (H(1))^H * (H(2))^H * \dots * (H(k))^H$ (for complex flavors)

as returned by p?gerqf . Q is of order m if $\text{side} = 'L'$ and of order n if $\text{side} = 'R'$.

Input Parameters

<i>side</i>	(global) CHARACTER.
= 'L':	apply Q or Q^T for real flavors (Q^H for complex flavors) from the left,
= 'R':	apply Q or Q^T for real flavors (Q^H for complex flavors) from the right.
<i>trans</i>	(global) CHARACTER.
= 'N':	apply Q (no transpose)
= 'T':	apply Q^T (transpose, for real flavors)
= 'C':	apply Q^H (conjugate transpose, for complex flavors)

m

(global) INTEGER.

The number of rows to be operated on, that is, the number of rows of the distributed submatrix $\text{sub}(C)$. $m \geq 0$.

n

(global) INTEGER.

The number of columns to be operated on, that is, the number of columns of the distributed submatrix $\text{sub}(C)$. $n \geq 0$.

k

(global) INTEGER.

The number of elementary reflectors whose product defines the matrix Q .

If $\text{side} = 'L'$, $m \geq k \geq 0$;

if $\text{side} = 'R'$, $n \geq k \geq 0$.

a

(local)

REAL for psormr2

DOUBLE PRECISION for pdormr2

COMPLEX for pcunmr2

COMPLEX*16 for pzunmr2.

Pointer into the local memory to an array, size

$(\text{lld}_a, \text{LOCc}(ja+m-1)$ if $\text{side}='L'$,

$(\text{lld}_a, \text{LOCc}(ja+n-1)$ if $\text{side}='R'$,

where $\text{lld}_a \geq \max(1, \text{LOCr}(ia+k-1))$.

On entry, the i -th row must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by [p?gerqf](#) in the k rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.

The argument $A(ia:ia+k-1, ja:*)$ is modified by the routine but restored on exit.

ia

(global) INTEGER.

The row index in the global array A indicating the first row of $\text{sub}(A)$.

ja

(global) INTEGER.

The column index in the global array A indicating the first column of $\text{sub}(A)$.

desca

(global and local) INTEGER array of size (*dlen_*). The array descriptor for the distributed matrix A .

tau

(local)

REAL for psormr2

DOUBLE PRECISION for pdormr2

COMPLEX for pcunmr2

COMPLEX*16 for pzunmr2.

Array, size $\text{LOCc}(ia+k-1)$. This array contains the scalar factors $\tau(i)$ of the elementary reflector $H(i)$, as returned by [p?gerqf](#). This array is tied to the distributed matrix A .

c	(local) REAL for psormr2 DOUBLE PRECISION for pdormr2 COMPLEX for pcunmr2 COMPLEX*16 for pzunmr2. Pointer into the local memory to an array, size($l1d_c$, $\text{LOCc}(jc+n-1)$). On entry, the local pieces of the distributed matrix sub (C).
ic	(global) INTEGER. The row index in the global array C indicating the first row of sub(C).
jc	(global) INTEGER. The column index in the global array C indicating the first column of sub(C).
desc_C	(global and local) INTEGER array of size ($dlen_$). The array descriptor for the distributed matrix C .
$work$	(local) REAL for psormr2 DOUBLE PRECISION for pdormr2 COMPLEX for pcunmr2 COMPLEX*16 for pzunmr2. Workspace array, size ($lwork$).
$lwork$	(local or global) INTEGER. The dimension of the array $work$. $lwork$ is local input and must be at least $\text{if } side = 'L', lwork \geq mpc0 + \max(\max(1, nqc0), \text{numroc}(\text{numroc}(m + iroffc, mb_a, 0, 0, nprow), mb_a, 0, 0, lcm)),$ $\text{if } side = 'R', lwork \geq nqc0 + \max(1, mpc0),$ where $lcm = lcm(nprow, npcol),$ $irow = \text{indxg2p}(ic, mb_c, myrow, rsrc_c, nprow),$ $icol = \text{indxg2p}(jc, nb_c, mycol, csrc_c, npcol),$ $Mpc0 = \text{numroc}(m + iroffc, mb_c, myrow, icrow, nprow),$ $Nqc0 = \text{numroc}(n + icoffc, nb_c, mycol, iccol, npcol),$

`ilcm`, `indxg2p` and `numroc` are ScaLAPACK tool functions; `myrow`, `mycol`, `nprov`, and `ncol` can be determined by calling the subroutine `blacs_gridinfo`.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pixerbla`.

Output Parameters

<code>c</code>	On exit, <code>c</code> is overwritten by $Q^* \text{sub}(C)$, or $Q^T * \text{sub}(C) / Q^H * \text{sub}(C)$, or $\text{sub}(C) * Q$, or $\text{sub}(C) * Q^T / \text{sub}(C) * Q^H$
<code>work</code>	On exit, <code>work(1)</code> returns the minimal and optimal <code>lwork</code> .
<code>info</code>	(local) INTEGER. = 0: successful exit < 0: if the i -th argument is an array and the j -entry had an illegal value, then <code>info = - (i*100+j)</code> , if the i -th argument is a scalar and had an illegal value, then <code>info = -i</code> .

NOTE

The distributed submatrices `A(ia:*, ja:*)` and `C(ic:ic+m-1, jc:jc+n-1)` must verify some alignment properties, namely the following expressions should be true:

If `side = 'L'`, (`nb_a.eq.mb_c .AND. icooffa.eq.iroffc`)
If `side = 'R'`, (`nb_a.eq.nb_c .AND. icooffa.eq.icooffc .AND. iacol.eq.iccol`).

p?pbtrs

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by p?pbtrf.

Syntax

Fortran:

```
call pspbtrs(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

```
call pdpbtrs(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

```
call pcpbtrs(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

```
call pzpbtrs(uplo, trans, n, bw, nrhs, a, ja, desca, b, ib, descb, af, laf, work, lwork, info)
```

C:

```

void pspbtrs (char *uplo , char *trans , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs ,
float *a , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb ,
float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pdpbtrs (char *uplo , char *trans , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs ,
double *a , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb ,
double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcpbtrs (char *uplo , char *trans , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzpbtrs (char *uplo , char *trans , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );

```

Include Files

- C: mkl_scalapack.h

Description

The `p?pbtrs` routine solves a banded triangular system of linear equations

$A(1:n, ja:ja+n-1) * X = B(jb:jb+n-1, 1:nrhs)$

or

$A(1:n, ja:ja+n-1)^T * X = B(jb:jb+n-1, 1:nrhs)$ for real flavors,

$A(1:n, ja:ja+n-1)^H * X = B(jb:jb+n-1, 1:nrhs)$ for complex flavors,

where $A(1:n, ja:ja+n-1)$ is a banded triangular matrix factor produced by the Cholesky factorization code `p?pbtrf` and is stored in $A(1:n, ja:ja+n-1)$ and `af`. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to `uplo`.

Routine `p?pbtrf` must be called first.

Input Parameters

`uplo`

(global) CHARACTER. Must be 'U' or 'L'.

If `uplo` = 'U', upper triangle of $A(1:n, ja:ja+n-1)$ is stored;

If `uplo` = 'L', lower triangle of $A(1:n, ja:ja+n-1)$ is stored.

`trans`

(global) CHARACTER. Must be 'N' or 'T' or 'C'.

If `trans` = 'N', solve with $A(1:n, ja:ja+n-1)$;

If `trans` = 'T' or 'C' for real flavors, solve with $A(1:n, ja:ja+n-1)^T$.

If `trans` = 'C' for complex flavors, solve with conjugate transpose ($A(1:n, ja:ja+n-1)^H$).

`n`

(global) INTEGER.

The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$. $n \geq 0$.

`bw`

(global) INTEGER.

The number of subdiagonals in 'L' or 'U', $0 \leq bw \leq n-1$.

nrhs

(global) INTEGER.

The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs)$; $nrhs \geq 0$.

a

(local)

REAL for pspbtrsv

DOUBLE PRECISION for pdpbtrsv

COMPLEX for pcpbtrsv

COMPLEX*16 for pzpbtrsv.

Pointer into the local memory to an array with the first size $lld_a \geq (bw + 1)$, stored in *desca*.

On entry, this array contains the local pieces of the n -by- n symmetric banded distributed Cholesky factor L or $L^T * A(1:n, ja:ja+n-1)$.

This local portion is stored in the packed banded format used in LAPACK. See the *Application Notes* below and the ScaLAPACK manual for more detail on the format of distributed matrices.

ja

(global) INTEGER. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desca

(global and local) INTEGER array, size (*dlen*). The array descriptor for the distributed matrix *A*.

If 1D type (*dtype_a* = 501), then *dlen* ≥ 7 ;

If 2D type (*dtype_a* = 1), then *dlen* ≥ 9 .

Contains information on mapping of *A* to memory. (See ScaLAPACK manual for full description and options.)

b

(local)

REAL for pspbtrsv

DOUBLE PRECISION for pdpbtrsv

COMPLEX for pcpbtrsv

COMPLEX*16 for pzpbtrsv.

Pointer into the local memory to an array of local lead size $lld_b \geq nb$.

On entry, this array contains the local pieces of the right hand sides $B(jb:jb+n-1, 1:nrhs)$.

ib

(global) INTEGER. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb

(global and local) INTEGER array, size (*dlen*). The array descriptor for the distributed matrix *B*.

If 1D type (*dtype_b* = 502), then *dlen* ≥ 7 ;

If 2D type (*dtype_b* = 1), then *dlen* \geq 9.

Contains information on mapping of *B* to memory. Please, see ScaLAPACK manual for full description and options.

laf

(local)

INTEGER. The size of user-input auxiliary Fillin space *af*. Must be *laf* \geq (*nb*+2**bw*) * *bw*. If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work

(local)

REAL for pspbtrsv

DOUBLE PRECISION for pdpbtrsv

COMPLEX for pcpbtrsv

COMPLEX*16 for pzpbtrsv.

The array *work* is a temporary workspace array of size *lwork*. This space may be overwritten in between calls to routines.

lwork

(local or global) INTEGER. The size of the user-input workspace *work*, must be at least *lwork* \geq *bw***nrhs*. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

af

(local)

REAL for pspbtrsv

DOUBLE PRECISION for pdpbtrsv

COMPLEX for pcpbtrsv

COMPLEX*16 for pzpbtrsv.

The array *af* is of size *laf*. It contains auxiliary Fillin space. Fillin is created during the factorization routine [p?pbtrf](#) and this is stored in *af*. If a linear system is to be solved using [p?pbtrs](#) after the factorization routine, *af* must not be altered after the factorization.

b

On exit, this array contains the local piece of the solutions distributed matrix *X*.

work(1)

On exit, *work*(1) contains the minimum value of *lwork*.

info

(local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,

then *info* = -(*i**100+*j*),

if the *i*-th argument is a scalar and had an illegal value,

then *info* = -*i*.

Application Notes

If the factorization routine and the solve routine are to be called separately to solve various sets of right-hand sides using the same coefficient matrix, the auxiliary space *af* must not be altered between calls to the factorization routine and the solve routine.

The best algorithm for solving banded and tridiagonal linear systems depends on a variety of parameters, especially the bandwidth. Currently, only algorithms designed for the case $N/P \gg bw$ are implemented. These algorithms go by many names, including Divide and Conquer, Partitioning, domain decomposition-type, etc.

The Divide and Conquer algorithm assumes the matrix is narrowly banded compared with the number of equations. In this situation, it is best to distribute the input matrix *A* one-dimensionally, with columns atomic and rows divided amongst the processes. The basic algorithm divides the banded matrix up into *P* pieces with one stored on each processor, and then proceeds in 2 phases for the factorization or 3 for the solution of a linear system.

1. **Local Phase:** The individual pieces are factored independently and in parallel. These factors are applied to the matrix creating fill-in, which is stored in a non-inspectable way in auxiliary space *af*. Mathematically, this is equivalent to reordering the matrix *A* as PAP^T and then factoring the principal leading submatrix of size equal to the sum of the sizes of the matrices factored on each processor. The factors of these submatrices overwrite the corresponding parts of *A* in memory.
2. **Reduced System Phase:** A small ($bw^*(P-1)$) system is formed representing interaction of the larger blocks and is stored (as are its factors) in the space *af*. A parallel Block Cyclic Reduction algorithm is used. For a linear system, a parallel front solve followed by an analogous backsolve, both using the structure of the factored matrix, are performed.
3. **Back Substitution Phase:** For a linear system, a local backsubstitution is performed on each processor in parallel.

p?pttrsV

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by p?pttrf .

Syntax

Fortran:

```
call pspttrsv(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork,
info)

call pdpttrsv(uplo, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work, lwork,
info)

call pcpttrsv(uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work,
lwork, info)

call pzpttrsv(uplo, trans, n, nrhs, d, e, ja, desca, b, ib, descb, af, laf, work,
lwork, info)
```

C:

```
void pspttrsv (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *d , float *e , MKL_INT
*ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *af , MKL_INT
*laf , float *work , MKL_INT *lwork , MKL_INT *info );

void pdpttrsv (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *d , double *e ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double *af ,
MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );
```

```
void pcptttrs (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , float *d ,
MKL_Complex8 *e , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT
*lwork , MKL_INT *info );

void pzptttrs (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , double *d ,
MKL_Complex16 *e , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib ,
MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT
*lwork , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?ptttrs routine solves a tridiagonal triangular system of linear equations

$A(1:n, ja:ja+n-1)^{*}X = B(jb:jb+n-1, 1:nrhs)$

or

$A(1:n, ja:ja+n-1)^T*X = B(jb:jb+n-1, 1:nrhs)$ for real flavors,

$A(1:n, ja:ja+n-1)^H*X = B(jb:jb+n-1, 1:nrhs)$ for complex flavors,

where $A(1:n, ja:ja+n-1)$ is a tridiagonal triangular matrix factor produced by the Cholesky factorization code p?pttrf and is stored in $A(1:n, ja:ja+n-1)$ and af. The matrix stored in $A(1:n, ja:ja+n-1)$ is either upper or lower triangular according to uplo.

Routine p?pttrf must be called first.

Input Parameters

<i>uplo</i>	(global) CHARACTER. Must be 'U' or 'L'.
	If <i>uplo</i> = 'U', upper triangle of $A(1:n, ja:ja+n-1)$ is stored;
	If <i>uplo</i> = 'L', lower triangle of $A(1:n, ja:ja+n-1)$ is stored.
<i>trans</i>	(global) CHARACTER. Must be 'N' or 'C'.
	If <i>trans</i> = 'N', solve with $A(1:n, ja:ja+n-1)$;
	If <i>trans</i> = 'C' (for complex flavors), solve with conjugate transpose ($A(1:n, ja:ja+n-1)^H$).
<i>n</i>	(global) INTEGER.
	The number of rows and columns to be operated on, that is, the order of the distributed submatrix $A(1:n, ja:ja+n-1)$. $n \geq 0$.
<i>nrhs</i>	(global) INTEGER.
	The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs)$; $nrhs \geq 0$.
<i>d</i>	(local)
	REAL for psptttrs
	DOUBLE PRECISION for pdptttrs
	COMPLEX for pcptttrs

COMPLEX*16 for `pzpttrs`.

Pointer to the local part of the global vector storing the main diagonal of the matrix; must be of size $\geq \text{desc}(nb_)$.

e

(local)

REAL for `pspttrs`

DOUBLE PRECISION for `pdpttrs`

COMPLEX for `pcpttrs`

COMPLEX*16 for `pzpttrs`.

Pointer to the local part of the global vector storing the upper diagonal of the matrix; must be of size $\geq \text{desc}(nb_)$. Globally, $du(n)$ is not referenced, and du must be aligned with d .

ja

(global) **INTEGER**. The index in the global array *A* that points to the start of the matrix to be operated on (which may be either all of *A* or a submatrix of *A*).

desc

(global and local) **INTEGER array**, size (*dlen_*). The array descriptor for the distributed matrix *A*.

If 1D type (*dtype_a* = 501 or 502), then *dlen* ≥ 7 ;

If 2D type (*dtype_a* = 1), then *dlen* ≥ 9 .

Contains information on mapping of *A* to memory. See ScaLAPACK manual for full description and options.

b

(local)

REAL for `pspttrs`

DOUBLE PRECISION for `pdpttrs`

COMPLEX for `pcpttrs`

COMPLEX*16 for `pzpttrs`.

Pointer into the local memory to an array of local lead size $lld_b \geq nb$.

On entry, this array contains the local pieces of the right hand sides $B(jb:jb+n-1, 1:nrhs)$.

ib

(global) **INTEGER**. The row index in the global array *B* that points to the first row of the matrix to be operated on (which may be either all of *B* or a submatrix of *B*).

descb

(global and local) **INTEGER array**, size (*dlen_*). The array descriptor for the distributed matrix *B*.

If 1D type (*dtype_b* = 502), then *dlen* ≥ 7 ;

If 2D type (*dtype_b* = 1), then *dlen* ≥ 9 .

Contains information on mapping of *B* to memory. See ScaLAPACK manual for full description and options.

laf

(local)

INTEGER. The size of user-input auxiliary Fillin space *af*. Must be $laf \geq (nb+2 * bw) * bw$.

If *laf* is not large enough, an error code will be returned and the minimum acceptable size will be returned in *af*(1).

work

(local)

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

The array *work* is a temporary workspace array of size *lwork*. This space may be overwritten in between calls to routines.

lwork

(local or global) INTEGER. The size of the user-input workspace *work*, must be at least $lwork \geq (10+2*\min(100, nrhs))*npcol+4*nrhs$. If *lwork* is too small, the minimal acceptable size will be returned in *work*(1) and an error code is returned.

Output Parameters

d, e

(local).

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

On exit, these arrays contain information on the factors of the matrix.

af

(local)

REAL for pspttrsv

DOUBLE PRECISION for pdpttrsv

COMPLEX for pcpttrsv

COMPLEX*16 for pzpttrsv.

The array *af* is of size *laf*. It contains auxiliary Fillin space. Fillin is created during the factorization routine [p?pbtrf](#) and this is stored in *af*. If a linear system is to be solved using [p?pttrs](#) after the factorization routine, *af* must not be altered after the factorization.

b

On exit, this array contains the local piece of the solutions distributed matrix *X*.

work(1)

On exit, *work*(1) contains the minimum value of *lwork*.

info

(local) INTEGER.

= 0: successful exit

< 0: if the *i*-th argument is an array and the *j*-entry had an illegal value,
then *info* = -(*i**100+*j*),

if the i -th argument is a scalar and had an illegal value,
then $\text{info} = -i$.

p?potf2

*Computes the Cholesky factorization of a symmetric/
Hermitian positive definite matrix (local unblocked
algorithm).*

Syntax

Fortran:

```
call pspotf2(uplo, n, a, ia, ja, desca, info)
call pdpotf2(uplo, n, a, ia, ja, desca, info)
call pcpotf2(uplo, n, a, ia, ja, desca, info)
call pzpotf2(uplo, n, a, ia, ja, desca, info)
```

C:

```
void pspotf2 (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , MKL_INT *info );
void pdpotf2 (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
void pcpotf2 (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
void pzpotf2 (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?potf2` routine computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite distributed matrix $\text{sub}(A) = A(\text{ia}: \text{ia}+n-1, \text{ja}: \text{ja}+n-1)$.

The factorization has the form

$\text{sub}(A) = U' * U$, if $\text{uplo} = 'U'$, or $\text{sub}(A) = L * L'$, if $\text{uplo} = 'L'$,

where U is an upper triangular matrix, L is lower triangular. X' denotes transpose (conjugate transpose) of X .

Input Parameters

<code>uplo</code>	(global) CHARACTER.
	Specifies whether the upper or lower triangular part of the symmetric/ Hermitian matrix A is stored.
	= ' <code>U</code> ': upper triangle of $\text{sub}(A)$ is stored;
	= ' <code>L</code> ': lower triangle of $\text{sub}(A)$ is stored.

`n` (global) INTEGER.

The number of rows and columns to be operated on, that is, the order of the distributed submatrix sub (A). $n \geq 0$.

a

(local)
 REAL for pspotf2
 DOUBLE PRECISION for pdpotf2
 COMPLEX for pcpotf2
 COMPLEX*16 for pzpotf2.

Pointer into the local memory to an array of size ($l1d_a$, $LOCc(ja+n-1)$) containing the local pieces of the n -by- n symmetric distributed matrix sub(A) to be factored.

If $uplo = 'U'$, the leading n -by- n upper triangular part of sub(A) contains the upper triangular matrix and the strictly lower triangular part of this matrix is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of sub(A) contains the lower triangular matrix and the strictly upper triangular part of sub(A) is not referenced.

ia, ja

(global) INTEGER.

The row and column indices in the global array A indicating the first row and the first column of the sub(A), respectively.

desca

(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix A .

Output Parameters

a

(local)
 On exit,
 if $uplo = 'U'$, the upper triangular part of the distributed matrix contains the Cholesky factor U ;
 if $uplo = 'L'$, the lower triangular part of the distributed matrix contains the Cholesky factor L .

info

(local) INTEGER.
 = 0: successful exit
 < 0: if the i -th argument is an array and the j -entry had an illegal value,
 then $info = -(i*100+j)$,
 if the i -th argument is a scalar and had an illegal value,
 then $info = -i$.
 > 0: if $info = k$, the leading minor of order k is not positive definite, and the factorization could not be completed.

p?rot

Applies a planar rotation to two distributed vectors.

Syntax

Fortran:

```
call psrot( n, x, ix, jx, descx, incx, y, iy, jy, descy, incy, cs, sn, work, lwork,
info )
call pdrot( n, x, ix, jx, descx, incx, y, iy, jy, descy, incy, cs, sn, work, lwork,
info )
```

Include Files

- C: mkl_scalapack.h

Description

p?rot applies a planar rotation defined by *cs* and *sn* to the two distributed vectors sub(*x*) and sub(*y*).

Input Parameters

<i>n</i>	(global) INTEGER	The number of elements to operate on when applying the planar rotation to <i>x</i> and <i>y</i> ($n \geq 0$).
<i>x</i>	REAL for psrot DOUBLE PRECISION for pdrot (local) array of dimension ($(jx-1)*descx(m_)$ + <i>ix</i> + ($n - 1$)*abs(<i>incx</i>))	This array contains the entries of the distributed vector sub(<i>x</i>).
<i>ix</i>	(global) INTEGER	The global row index of the submatrix of the distributed matrix <i>x</i> to operate on. If <i>incx</i> = 1, then it is required that <i>ix</i> = <i>iy</i> . $1 \leq ix \leq descx(m_)$.
<i>jx</i>	(global) INTEGER	The global column index of the submatrix of the distributed matrix <i>x</i> to operate on. If <i>incx</i> = <i>descx(m_)</i> , then it is required that <i>jx</i> = <i>jy</i> . $1 \leq ix \leq descx(n_)$.
<i>descx</i>	(global and local) INTEGER array of dimension 9	The array descriptor of the distributed matrix <i>x</i> .
<i>incx</i>	(global) INTEGER	The global increment for the elements of <i>x</i> . Only two values of <i>incx</i> are supported in this version, namely 1 and <i>descx(m_)</i> . Moreover, it must hold that <i>incx</i> = <i>descx(m_)</i> if <i>incy</i> = <i>descy(m_)</i> and that <i>incx</i> = 1 if <i>incy</i> = 1.
<i>y</i>	REAL for psrot DOUBLE PRECISION for pdrot (local) array of dimension ($(jy-1)*descy(m_)$ + <i>iy</i> + ($n - 1$)*abs(<i>incy</i>))	

This array contains the entries of the distributed vector $\text{sub}(y)$.

i_y

(global) INTEGER

The global row index of the submatrix of the distributed matrix *y* to operate on. If *incy* = 1, then it is required that *i_y* = *i_x*. $1 \leq i_y \leq \text{descy}(m_)$.

j_y

(global) INTEGER

The global column index of the submatrix of the distributed matrix *y* to operate on. If *incy* = $\text{descx}(m_)$, then it is required that *j_y* = *j_x*. $1 \leq j_y \leq \text{descy}(n_)$.

descy

(global and local) INTEGER array of dimension 9

The array descriptor of the distributed matrix *y*.

incy

(global) INTEGER

The global increment for the elements of *y*. Only two values of *incy* are supported in this version, namely 1 and $\text{descy}(m_)$. Moreover, it must hold that *incy* = $\text{descy}(m_)$ if *incx* = $\text{descx}(m_)$ and that *incy* = 1 if *incx* = 1.

cs, sn

(global)

REAL for psrot

DOUBLE PRECISION for pdrot

The parameters defining the properties of the planar rotation. It must hold that $0 \leq cs, sn \leq 1$ and that $sn^2 + cs^2 = 1$. The latter is hardly checked in finite precision arithmetics.

work

REAL for psrot

DOUBLE PRECISION for pdrot

(local workspace) array of dimension *lwork*

lwork

(local) INTEGER

The length of the workspace array *work*.

If *incx* = 1 and *incy* = 1, then *lwork* = $2 * \text{descx}(mb_)$

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by [pxerbla](#).

OUTPUT Parameters

x

y

work(1)

On exit, if *info* = 0, *work(1)* returns the optimal *lwork*

info

(global) INTEGER

= 0: successful exit

< 0: if $info = -i$, the i -th argument had an illegal value.

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*100+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

p?rscl

Multiples a vector by the reciprocal of a real scalar.

Syntax

Fortran:

```
call psrscl(n, sa, sx, ix, jx, descx, incx)
call pdrscl(n, sa, sx, ix, jx, descx, incx)
call pcsrcscl(n, sa, sx, ix, jx, descx, incx)
call pzdrscl(n, sa, sx, ix, jx, descx, incx)
```

C:

```
void psrscl (MKL_INT *n , float *sa , float *sx , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pdrscl (MKL_INT *n , double *sa , double *sx , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
void pcsrcscl (MKL_INT *n , float *sa , MKL_Complex8 *sx , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
void pzdrscl (MKL_INT *n , double *sa , MKL_Complex16 *sx , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
```

Include Files

- C: mkl_scalapack.h

Description

The p?rscl routine multiplies an n -element real/complex vector $\text{sub}(x)$ by the real scalar $1/a$. This is done without overflow or underflow as long as the final result $\text{sub}(x)/a$ does not overflow or underflow.

$\text{sub}(x)$ denotes $x(ix:ix+n-1, jx:jx)$, if $incx = 1$,

and $x(ix:ix, jx:jx+n-1)$, if $incx = m_x$.

Input Parameters

n (global) INTEGER.

The number of components of the distributed vector $\text{sub}(x)$. $n \geq 0$.

sa REAL for psrscl/pcsrcscl

DOUBLE PRECISION for pdrscl/pzdrscl.

The scalar a that is used to divide each component of the vector x . This parameter must be ≥ 0 .

sx REAL for psrscl

DOUBLE PRECISION for pdrscl
 COMPLEX for pcsrcsl
 COMPLEX*16 for pzdrscl.

Array containing the local pieces of a distributed matrix of size of at least $((j_x-1)*m_x + i_x + (n-1)*\text{abs}(incx))$. This array contains the entries of the distributed vector sub(x).

i_x (global) INTEGER. The row index of the submatrix of the distributed matrix X to operate on.

j_x (global) INTEGER. The column index of the submatrix of the distributed matrix X to operate on.

descx (global and local) INTEGER. Array of size 9. The array descriptor for the distributed matrix X .

incx (global) INTEGER. The increment for the elements of X . This version supports only two values of *incx*, namely 1 and *m_x*.

Output Parameters

s_x On exit, the result x/a .

p?sygs2/p?hegs2

Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf (local unblocked algorithm).

Syntax

Fortran:

```
call pssygs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pdsygs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pchegs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
call pzhegs2(ibtype, uplo, n, a, ia, ja, desca, b, ib, jb, descb, info)
```

C:

```
void pssygs2 (MKL_INT *ibtype , char *uplo , MKL_INT *n , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
MKL_INT *info );
void pdsygs2 (MKL_INT *ibtype , char *uplo , MKL_INT *n , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *info );
```

```
void pchegs2 (MKL_INT *ibtype , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );
void pzhegs2 (MKL_INT *ibtype , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb ,
MKL_INT *descb , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?sygs2/p?hegs2 routine reduces a real symmetric-definite or a complex Hermitian positive-definite generalized eigenproblem to standard form.

Here $\text{sub}(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$, and $\text{sub}(B)$ denotes $B(ib:ib+n-1, jb:jb+n-1)$.

If $ibtype = 1$, the problem is

$$\text{sub}(A) * x = \lambda * \text{sub}(B) * x$$

and $\text{sub}(A)$ is overwritten by

$\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$ or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$ - for real flavors, and
 $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$ or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$ - for complex flavors.

If $ibtype = 2$ or 3 , the problem is

$$\text{sub}(A) * \text{sub}(B) * x = \lambda * x \text{ or } \text{sub}(B) * \text{sub}(A) * x = \lambda * x$$

and $\text{sub}(A)$ is overwritten by

$U * \text{sub}(A) * U^T$ or $L^{**} T * \text{sub}(A) * L$ - for real flavors and
 $U * \text{sub}(A) * U^H$ or $L^{**} H * \text{sub}(A) * L$ - for complex flavors.

The matrix $\text{sub}(B)$ must have been previously factorized as $U^T * U$ or $L * L^T$ (for real flavors), or as $U^H * U$ or $L * L^H$ (for complex flavors) by p?potrf.

Input Parameters

<i>ibtype</i>	(global) INTEGER. = 1: compute $\text{inv}(U^T) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^T)$ for real subroutines, and $\text{inv}(U^H) * \text{sub}(A) * \text{inv}(U)$, or $\text{inv}(L) * \text{sub}(A) * \text{inv}(L^H)$ for complex subroutines; = 2 or 3: compute $U * \text{sub}(A) * U^T$, or $L^T * \text{sub}(A) * L$ for real subroutines, and $U * \text{sub}(A) * U^H$ or $L^H * \text{sub}(A) * L$ for complex subroutines.
<i>uplo</i>	(global) CHARACTER Specifies whether the upper or lower triangular part of the symmetric/ Hermitian matrix $\text{sub}(A)$ is stored, and how $\text{sub}(B)$ is factorized. = 'U': Upper triangular of $\text{sub}(A)$ is stored and $\text{sub}(B)$ is factorized as $U^T * U$ (for real subroutines) or as $U^H * U$ (for complex subroutines).

= 'L': Lower triangular of sub(A) is stored and sub(B) is factorized as L^*L^T (for real subroutines) or as L^*L^H (for complex subroutines)

n (global) INTEGER.

The order of the matrices sub(A) and sub(B). $n \geq 0$.

a (local)

REAL for pssygs2

DOUBLE PRECISION for pdsygs2

COMPLEX for pchegs2

COMPLEX*16 for pzhegs2.

Pointer into the local memory to an array, size (lld_a , $LOCc(ja+n-1)$).

On entry, this array contains the local pieces of the n -by- n symmetric/Hermitian distributed matrix sub(A).

If $uplo = 'U'$, the leading n -by- n upper triangular part of sub(A) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(A) is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of sub(A) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(A) is not referenced.

ia, ja

(global) INTEGER.

The row and column indices in the global array A indicating the first row and the first column of the sub(A), respectively.

desca

(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix A .

B

(local)

REAL for pssygs2

DOUBLE PRECISION for pdsygs2

COMPLEX for pchegs2

COMPLEX*16 for pzhegs2.

Pointer into the local memory to an array, size (lld_b , $LOCc(jb+n-1)$).

On entry, this array contains the local pieces of the triangular factor from the Cholesky factorization of sub(B) as returned by [p?potrf](#).

ib, jb

(global) INTEGER.

The row and column indices in the global array B indicating the first row and the first column of the sub(B), respectively.

descb

(global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix B .

Output Parameters

a

(local)

On exit, if $info = 0$, the transformed matrix is stored in the same format as $\text{sub}(A)$.

info INTEGER.
 $= 0$: successful exit.
 < 0 : if the i -th argument is an array and the j -entry had an illegal value,
then $info = - (i * 100)$,
if the i -th argument is a scalar and had an illegal value,
then $info = -i$.

p?sytd2/p?hetd2

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).

Syntax

Fortran:

```
call pssytd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pdsytd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pchetd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
call pzhetd2(uplo, n, a, ia, ja, desca, d, e, tau, work, lwork, info)
```

Include Files

- C: mkl_scalapack.h

Description

The p?sytd2/p?hetd2 routine reduces a real symmetric/complex Hermitian matrix $\text{sub}(A)$ to symmetric/Hermitian tridiagonal form T by an orthogonal/unitary similarity transformation:

$$Q^* \text{sub}(A) * Q = T, \text{ where } \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1).$$

Input Parameters

uplo (global) CHARACTER.
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix $\text{sub}(A)$ is stored:
 $= 'U'$: upper triangular
 $= 'L'$: lower triangular

n (global) INTEGER.
The number of rows and columns to be operated on, that is, the order of the distributed submatrix $\text{sub}(A)$. $n \geq 0$.

a (local)
REAL for pssytd2
DOUBLE PRECISION for pdsytd2

COMPLEX for `pchetd2`
 COMPLEX*16 for `pzhetd2`.
 Pointer into the local memory to an array, size (`lld_a`, `LOCc(ja+n-1)`).
 On entry, this array contains the local pieces of the n -by- n symmetric/
 Hermitian distributed matrix sub(A).
 If `uplo` = 'U', the leading n -by- n upper triangular part of sub(A) contains
 the upper triangular part of the matrix, and the strictly lower triangular part
 of sub(A) is not referenced.
 If `uplo` = 'L', the leading n -by- n lower triangular part of sub(A) contains
 the lower triangular part of the matrix, and the strictly upper triangular part
 of sub(A) is not referenced.

`ia, ja`
 (global) INTEGER.
 The row and column indices in the global array A indicating the first row
 and the first column of the sub(A), respectively.

`desca`
 (global and local) INTEGER array, size (`dlen_`). The array descriptor for the
 distributed matrix A .

`work`
 (local)
 REAL for `pssytd2`
 DOUBLE PRECISION for `pdsytd2`
 COMPLEX for `pchetd2`
 COMPLEX*16 for `pzhetd2`.
 The array `work` is a temporary workspace array of size `/work`.

Output Parameters

`a`
 On exit, if `uplo` = 'U', the diagonal and first superdiagonal of sub(A) are
 overwritten by the corresponding elements of the tridiagonal matrix T , and
 the elements above the first superdiagonal, with the array `tau`, represent
 the orthogonal/unitary matrix Q as a product of elementary reflectors;
 if `uplo` = 'L', the diagonal and first subdiagonal of A are overwritten by
 the corresponding elements of the tridiagonal matrix T , and the elements
 below the first subdiagonal, with the array `tau`, represent the orthogonal/
 unitary matrix Q as a product of elementary reflectors. See the *Application
 Notes* below.

`d`
 (local)
 REAL for `pssytd2/pchetd2`
 DOUBLE PRECISION for `pdsytd2/pzhetd2`.
 Array, size (`LOCc(ja+n-1)`). The diagonal elements of the tridiagonal
 matrix T :
 $d(i) = a(i,i); d$ is tied to the distributed matrix A .

`e`
 (local)
 REAL for `pssytd2/pchetd2`

DOUBLE PRECISION for `pdsytd2/pzhetd2`.

`Array, size(LOCc(ja+n-1))`,

if `uplo = 'U'`, `LOCc(ja+n-2)` otherwise.

The off-diagonal elements of the tridiagonal matrix T :

$e(i) = a(i,i+1)$ if `uplo = 'U'`,

$e(i) = a(i+1,i)$ if `uplo = 'L'`.

e is tied to the distributed matrix A .

`tau`

(local)

REAL for `pssytd2`

DOUBLE PRECISION for `pdsytd2`

COMPLEX for `pchetd2`

COMPLEX*16 for `pzhetd2`.

`Array, size(LOCc(ja+n-1))`.

The scalar factors of the elementary reflectors. τ is tied to the distributed matrix A .

`work(1)`

On exit, `work(1)` returns the minimal and optimal value of `lwork`.

`lwork`

(local or global) INTEGER.

The dimension of the workspace array `work`.

`lwork` is local input and must be at least $lwork \geq 3n$.

If `lwork = -1`, then `lwork` is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

`info`

(local) INTEGER.

= 0: successful exit

< 0: if the i -th argument is an array and the j -entry had an illegal value,
then $info = -(i*100)$,

if the i -th argument is a scalar and had an illegal value,

then $info = -i$.

Application Notes

If `uplo = 'U'`, the matrix Q is represented as a product of elementary reflectors

$$Q = H(n-1) * \dots * H(2) * H(1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T,$$

where τ is a real/complex scalar, and v is a real/complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and τ in $TAU(ja+i-1)$.

If `uplo = 'L'`, the matrix Q is represented as a product of elementary reflectors

$Q = H(1) * H(2) * \dots * H(n-1)$.

Each $H(i)$ has the form

$H(i) = I - tau * v * v'$,

where tau is a real/complex scalar, and v is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and tau in $TAU(ja+i-1)$.

The contents of sub (A) on exit are illustrated by the following examples with $n = 5$:

$$\begin{array}{c} \text{if } uplo='U': \\ \left[\begin{array}{ccccc} d & e & v_2 & v_3 & v_4 \\ & d & e & v_3 & v_4 \\ & & d & e & v_4 \\ & & & d & e \\ & & & & d \end{array} \right] \\ \\ \text{if } uplo='L': \\ \left[\begin{array}{ccccc} d & & & & \\ e & d & & & \\ v_1 & e & d & & \\ v_1 & v_2 & e & d & \\ v_1 & v_2 & v_3 & e & d \end{array} \right] \end{array}$$

where d and e denotes diagonal and off-diagonal elements of T , and v_i denotes an element of the vector defining $H(i)$.

NOTE

The distributed submatrix sub(A) must verify some alignment properties, namely the following expression should be true:

($mb_a.eq.nb_a .AND. iroffa.eq.icoffa$) with $iroffa = \text{mod}(ia - 1, mb_a)$ and $icoffa = \text{mod}(ja - 1, nb_a)$.

p?trord

Reorders the Schur factorization of a general matrix.

Syntax

Fortran:

```
call pstrord( compq, select, para, n, t, it, jt, desct, q, iq, jq, descq, wr, wi, m,
work, lwork, iwork, liwork, info )
call pdtrord( compq, select, para, n, t, it, jt, desct, q, iq, jq, descq, wr, wi, m,
work, lwork, iwork, liwork, info )
```

Include Files

- C: mkl_scalapack.h

Description

p?trord reorders the real Schur factorization of a real matrix $A = Q * T^* Q^T$, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T , and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace.

T must be in Schur form (as returned by p?lahqr), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks.

This subroutine uses a delay and accumulate procedure for performing the off-diagonal updates.

Input Parameters

<i>compg</i>	(global) CHARACTER*1 = 'V': update the matrix q of Schur vectors; = 'N': do not update q .												
<i>select</i>	(global) INTEGER array, dimension (n) <i>select</i> specifies the eigenvalues in the selected cluster. To select a real eigenvalue $w(j)$, <i>select(j)</i> must be set to 1. To select a complex conjugate pair of eigenvalues $w(j)$ and $w(j+1)$, corresponding to a 2-by-2 diagonal block, either <i>select(j)</i> or <i>select(j+1)</i> or both must be set to 1; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.												
<i>para</i>	(global) INTEGER*6 Block parameters: <table border="0"> <tr> <td><i>para(1)</i></td><td>maximum number of concurrent computational windows allowed in the algorithm; $0 < para(1) \leq \min(nprow, ncol)$ must hold;</td></tr> <tr> <td><i>para(2)</i></td><td>number of eigenvalues in each window; $0 < para(2) < para(3)$ must hold;</td></tr> <tr> <td><i>para(3)</i></td><td>window size; $para(2) < para(3) < desc(m_b)$ must hold;</td></tr> <tr> <td><i>para(4)</i></td><td>minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; $0 \leq para(4) \leq 100$ must hold;</td></tr> <tr> <td><i>para(5)</i></td><td>width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; $0 < para(5) \leq desc(m_b)$ must hold.</td></tr> <tr> <td><i>para(6)</i></td><td>the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size; $0 < para(6) \leq para(2)$ must hold.</td></tr> </table>	<i>para(1)</i>	maximum number of concurrent computational windows allowed in the algorithm; $0 < para(1) \leq \min(nprow, ncol)$ must hold;	<i>para(2)</i>	number of eigenvalues in each window; $0 < para(2) < para(3)$ must hold;	<i>para(3)</i>	window size; $para(2) < para(3) < desc(m_b)$ must hold;	<i>para(4)</i>	minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; $0 \leq para(4) \leq 100$ must hold;	<i>para(5)</i>	width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; $0 < para(5) \leq desc(m_b)$ must hold.	<i>para(6)</i>	the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size; $0 < para(6) \leq para(2)$ must hold.
<i>para(1)</i>	maximum number of concurrent computational windows allowed in the algorithm; $0 < para(1) \leq \min(nprow, ncol)$ must hold;												
<i>para(2)</i>	number of eigenvalues in each window; $0 < para(2) < para(3)$ must hold;												
<i>para(3)</i>	window size; $para(2) < para(3) < desc(m_b)$ must hold;												
<i>para(4)</i>	minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; $0 \leq para(4) \leq 100$ must hold;												
<i>para(5)</i>	width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; $0 < para(5) \leq desc(m_b)$ must hold.												
<i>para(6)</i>	the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size; $0 < para(6) \leq para(2)$ must hold.												
<i>n</i>	(global) INTEGER The order of the globally distributed matrix T . $n \geq 0$.												
<i>t</i>	REAL for pstrord DOUBLE PRECISION for pdtrord (local) array, dimension ($\text{lld}_t, \text{LOC}_c(n)$). The local pieces of the global distributed upper quasi-triangular matrix T , in Schur form.												
<i>it, jt</i>	(global) INTEGER												

The row and column index in the global array t indicating the first column of T . $it = jt = 1$ must hold (see Application Notes).

desc_t (global and local) INTEGER array of dimension *dlen_*.

The array descriptor for the global distributed matrix t .

q REAL for pstrord

DOUBLE PRECISION for pdtrord

(local) array, dimension ($\text{ld}_q, \text{LOC}_c(n)$).

On entry, if $\text{comp}_q = 'V'$, the local pieces of the global distributed matrix Q of Schur vectors.

If $\text{comp}_q = 'N'$, *q* is not referenced.

iq, jq (global) INTEGER

The column index in the global array *q* indicating the first column of Q . $iq = jq = 1$ must hold (see Application Notes).

desc_q (global and local) INTEGER array of dimension *dlen_*.

The array descriptor for the global distributed matrix *q*.

work REAL for pstrord

DOUBLE PRECISION for pdtrord

(local workspace) array, dimension (*lwork*)

lwork (local) INTEGER

The dimension of the array *work*.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued by [pxerbla](#).

iwork (local workspace) INTEGER array, dimension (*liwork*)

liwork (local) INTEGER

The dimension of the array *iwork*.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued by [pxerbla](#)

OUTPUT Parameters

select (global) INTEGER array, dimension (*n*)

The (partial) reordering is displayed.

<i>t</i>	On exit, <i>t</i> is overwritten by the local pieces of the reordered matrix <i>T</i> , again in Schur form, with the selected eigenvalues in the globally leading diagonal blocks.
<i>q</i>	On exit, if <i>compq</i> = 'V', <i>q</i> has been postmultiplied by the global orthogonal transformation matrix which reorders <i>t</i> ; the leading <i>m</i> columns of <i>q</i> form an orthonormal basis for the specified invariant subspace. If <i>compq</i> = 'N', <i>q</i> is not referenced.
<i>wr, wi</i>	REAL for pstrord DOUBLE PRECISION for pdtrord (global) array, dimension (<i>n</i>) The real and imaginary parts, respectively, of the reordered eigenvalues of <i>t</i> . The eigenvalues are in principle stored in the same order as on the diagonal of <i>t</i> , with <i>wr</i> (<i>i</i>) = <i>t</i> (<i>i,i</i>) and, if <i>t</i> (<i>i:i+1,i:i+1</i>) is a 2-by-2 diagonal block, <i>wi</i> (<i>i</i>) > 0 and <i>wi</i> (<i>i+1</i>) = - <i>wi</i> (<i>i</i>). Note also that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.
<i>m</i>	(global) INTEGER The dimension of the specified invariant subspace. $0 \leq m \leq n$.
<i>work(1)</i>	On exit, if <i>info</i> = 0, <i>work(1)</i> returns the optimal <i>lwork</i> .
<i>iwork(1)</i>	On exit, if <i>info</i> = 0, <i>iwork(1)</i> returns the optimal <i>liwork</i> .
<i>info</i>	(global) INTEGER = 0: successful exit < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value. If the <i>i</i> -th argument is an array and the <i>j</i> -th entry had an illegal value, then <i>info</i> = -(<i>i</i> *1000+ <i>j</i>), if the <i>i</i> -th argument is a scalar and had an illegal value, then <i>info</i> = - <i>i</i> . > 0: here we have several possibilities <ul style="list-style-type: none">• Reordering of <i>t</i> failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); <i>t</i> may have been partially reordered, and <i>wr</i> and <i>wi</i> contain the eigenvalues in the same order as in <i>t</i>. On exit, <i>info</i> = {the index of <i>t</i> where the swap failed}.• A 2-by-2 block to be reordered split into two 1-by-1 blocks and the second block failed to swap with an adjacent block. On exit, <i>info</i> = {the index of <i>t</i> where the swap failed}.• If <i>info</i> = <i>n</i>+1, there is no valid BLACS context (see the BLACS documentation for details).

Application Notes

The following alignment requirements must hold:

- $\text{desc}(\text{mb}_\perp) = \text{desc}(\text{nb}_\perp) = \text{descq}(\text{mb}_\perp) = \text{descq}(\text{nb}_\perp)$
- $\text{desc}(\text{rsr}_\perp) = \text{descq}(\text{rsr}_\perp)$
- $\text{desc}(\text{csr}_\perp) = \text{descq}(\text{csr}_\perp)$

All matrices must be blocked by a block factor larger than or equal to two (3). This is to simplify reordering across processor borders in the presence of 2-by-2 blocks.

This algorithm cannot work on submatrices of t and q , i.e., $it = jt = iq = jq = 1$ must hold. This is however no limitation since [p?lahqr](#) does not compute Schur forms of submatrices anyway.

Parallel execution recommendations:

- Use a square grid, if possible, for maximum performance. The block parameters in *para* should be kept well below the data distribution block size.
- In general, the parallel algorithm strives to perform as much work as possible without crossing the block borders on the main block diagonal.

p?trsen

Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.

Syntax

Fortran:

```
call pstrsen( job, compq, select, para, n, t, it, jt, desc, q, iq, jq, descq, wr,
              wi, m, s, sep, work, lwork, iwork, liwork, info )
call pdtrsen( job, compq, select, para, n, t, it, jt, desc, q, iq, jq, descq, wr,
              wi, m, s, sep, work, lwork, iwork, liwork, info )
```

Include Files

- C: mkl_scalapack.h

Description

[p?trsen](#) reorders the real Schur factorization of a real matrix $A = Q*T^TQ^T$, so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix T , and the leading columns of Q form an orthonormal basis of the corresponding right invariant subspace. The reordering is performed by [p?trord](#).

Optionally the routine computes the reciprocal condition numbers of the cluster of eigenvalues and/or the invariant subspace.

T must be in Schur form (as returned by [p?lahqr](#)), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks.

Input Parameters

<i>job</i>	(global) CHARACTER*1
	Specifies whether condition numbers are required for the cluster of eigenvalues (<i>s</i>) or the invariant subspace (<i>sep</i>):
= 'N':	no condition numbers are required;
= 'E':	only the condition number for the cluster of eigenvalues is computed (<i>s</i>);

= 'V': only the condition number for the invariant subspace is computed (*sep*);
 = 'B': condition numbers for both the cluster and the invariant subspace are computed (*s* and *sep*).

compq

(global) CHARACTER*1
 = 'V': update the matrix *q* of Schur vectors;
 = 'N': do not update *q*.

select

(global) LOGICAL array, dimension (*n*)

select specifies the eigenvalues in the selected cluster. To select a real eigenvalue *w(j)*, *select(j)* must be set to .TRUE.. To select a complex conjugate pair of eigenvalues *w(j)* and *w(j+1)*, corresponding to a 2-by-2 diagonal block, either *select(j)* or *select(j+1)* or both must be set to .TRUE.; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.

para

(global) INTEGER*6

Block parameters:

<i>para(1)</i>	maximum number of concurrent computational windows allowed in the algorithm; $0 < \text{para}(1) \leq \min(\text{NPROW}, \text{NPCOL})$ must hold;
<i>para(2)</i>	number of eigenvalues in each window; $0 < \text{para}(2) < \text{para}(3)$ must hold;
<i>para(3)</i>	window size; $\text{para}(2) < \text{para}(3) < \text{desc}(mb_)$ must hold;
<i>para(4)</i>	minimal percentage of flops required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; $0 \leq \text{para}(4) \leq 100$ must hold;
<i>para(5)</i>	width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; $0 < \text{para}(5) \leq \text{desc}(mb_)$ must hold.
<i>para(6)</i>	the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size $0 < \text{para}(6) \leq \text{para}(2)$ must hold;

n

(global) INTEGER

The order of the globally distributed matrix *t*. $n \geq 0$.

t

REAL for pstrsen

DOUBLE PRECISION for pdtrsen

(local) array, dimension (*lld_t,LOC_c(n)*).

The local pieces of the global distributed upper quasi-triangular matrix T , in Schur form.

it, jt

(global) INTEGER

The row and column index in the global array t indicating the first column of T . $it = jt = 1$ must hold (see Application Notes).

$desc_t$

(global and local) INTEGER array of dimension $dlen_$.

The array descriptor for the global distributed matrix t .

q

REAL for pstrsen

DOUBLE PRECISION for pdtrsen

(local) array, dimension ($\text{lld}_q, \text{LOC}_c(n)$).

On entry, if $\text{compq} = 'V'$, the local pieces of the global distributed matrix Q of Schur vectors.

If $\text{compq} = 'N'$, q is not referenced.

iq, jq

(global) INTEGER

The column index in the global array q indicating the first column of Q . $iq = jq = 1$ must hold (see Application Notes).

$desc_q$

(global and local) INTEGER array of dimension $dlen_$.

The array descriptor for the global distributed matrix q .

$work$

REAL for pstrsen

DOUBLE PRECISION for pdtrsen

(local workspace) array, dimension ($lwork$)

$lwork$

(local) INTEGER

The dimension of the array $work$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued by [pxerbla](#).

$iwork$

(local workspace) INTEGER array, dimension ($liwork$)

$liwork$

(local) INTEGER

The dimension of the array $iwork$.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $iwork$ array, returns this value as the first entry of the $iwork$ array, and no error message related to $liwork$ is issued by [pxerbla](#).

OUTPUT Parameters

<i>t</i>	<i>t</i> is overwritten by the local pieces of the reordered matrix T , again in Schur form, with the selected eigenvalues in the globally leading diagonal blocks.
<i>q</i>	On exit, if <i>compq</i> = 'V', <i>q</i> has been postmultiplied by the global orthogonal transformation matrix which reorders <i>t</i> ; the leading <i>m</i> columns of <i>q</i> form an orthonormal basis for the specified invariant subspace. If <i>compq</i> = 'N', <i>q</i> is not referenced.
<i>wr, wi</i>	REAL for pstrsen DOUBLE PRECISION for pdtrsen (global) array, dimension (<i>n</i>) The real and imaginary parts, respectively, of the reordered eigenvalues of <i>t</i> . The eigenvalues are in principle stored in the same order as on the diagonal of <i>t</i> , with <i>wr</i> (<i>i</i>) = <i>t</i> (<i>i,i</i>) and, if <i>t</i> (<i>i:i+1,i:i+1</i>) is a 2-by-2 diagonal block, <i>wi</i> (<i>i</i>) > 0 and <i>wi</i> (<i>i+1</i>) = - <i>wi</i> (<i>i</i>). Note also that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.
<i>m</i>	(global) INTEGER The dimension of the specified invariant subspace. $0 \leq m \leq n$.
<i>s</i>	REAL for pstrsen DOUBLE PRECISION for pdtrsen (global) If <i>job</i> = 'E' or 'B', <i>s</i> is a lower bound on the reciprocal condition number for the selected cluster of eigenvalues. <i>s</i> cannot underestimate the true reciprocal condition number by more than a factor of $\text{sqrt}(n)$. If <i>m</i> = 0 or <i>n</i> , <i>s</i> = 1. If <i>job</i> = 'N' or 'V', <i>s</i> is not referenced.
<i>sep</i>	REAL for pstrsen DOUBLE PRECISION for pdtrsen (global) If <i>job</i> = 'V' or 'B', <i>sep</i> is the estimated reciprocal condition number of the specified invariant subspace. If <i>m</i> = 0 or <i>n</i> , <i>sep</i> = <i>norm(t)</i> . If <i>job</i> = 'N' or 'E', <i>sep</i> is not referenced.
<i>work(1)</i>	On exit, if <i>info</i> = 0, <i>work(1)</i> returns the optimal <i>lwork</i> .
<i>iwork(1)</i>	On exit, if <i>info</i> = 0, <i>iwork(1)</i> returns the optimal <i>liwork</i> .
<i>info</i>	(global) INTEGER = 0: successful exit

< 0: if $info = -i$, the i -th argument had an illegal value.

If the i -th argument is an array and the j -th entry had an illegal value, then $info = -(i*1000+j)$, if the i -th argument is a scalar and had an illegal value, then $info = -i$.

> 0: here we have several possibilities

- Reordering of t failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); t may have been partially reordered, and wr and wi contain the eigenvalues in the same order as in t .

On exit, $info = \{\text{the index of } t \text{ where the swap failed}\}$.

- A 2-by-2 block to be reordered split into two 1-by-1 blocks and the second block failed to swap with an adjacent block.

On exit, $info = \{\text{the index of } t \text{ where the swap failed}\}$.

- If $info = n+1$, there is no valid BLACS context (see the BLACS documentation for details).

Application Notes

The following alignment requirements must hold:

- $desc(m_b) = desc(t) = descq(m_b) = descq(n_b)$
- $desc(r_src) = descq(r_src)$
- $desc(c_src) = descq(c_src)$

All matrices must be blocked by a block factor larger than or equal to two (3). This to simplify reordering across processor borders in the presence of 2-by-2 blocks.

This algorithm cannot work on submatrices of t and q , i.e., $it = jt = iq = jq = 1$ must hold. This is however no limitation since [p?lahqr](#) does not compute Schur forms of submatrices anyway.

For parallel execution, use a square grid, if possible, for maximum performance. The block parameters in $para$ should be kept well below the data distribution block size.

In general, the parallel algorithm strives to perform as much work as possible without crossing the block borders on the main block diagonal.

p?trti2

Computes the inverse of a triangular matrix (local unblocked algorithm).

Syntax

Fortran:

```
call pstrti2(uplo, diag, n, a, ia, ja, desca, info)
call pdtrti2(uplo, diag, n, a, ia, ja, desca, info)
call pctrti2(uplo, diag, n, a, ia, ja, desca, info)
call pztrti2(uplo, diag, n, a, ia, ja, desca, info)
```

C:

```
void pstrti2 (char *uplo , char *diag , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , MKL_INT *info );
```

```
void pdtrti2 (char *uplo , char *diag , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , MKL_INT *info );
void pctrti2 (char *uplo , char *diag , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
void pztrti2 (char *uplo , char *diag , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The p?trti2 routine computes the inverse of a real/complex upper or lower triangular block matrix sub (A) = $A(ia:ia+n-1, ja:ja+n-1)$.

This matrix should be contained in one and only one process memory space (local operation).

Input Parameters

uplo

(global) CHARACTER*1.

Specifies whether the matrix sub (A) is upper or lower triangular.

= 'U': sub (A) is upper triangular

= 'L': sub (A) is lower triangular.

diag

(global) CHARACTER*1.

Specifies whether or not the matrix A is unit triangular.

= 'N': sub (A) is non-unit triangular

= 'U': sub (A) is unit triangular.

n

(global) INTEGER.

The number of rows and columns to be operated on, i.e., the order of the distributed submatrix sub (A). $n \geq 0$.

a

(local)

REAL for pstrti2

DOUBLE PRECISION for pdtrti2

COMPLEX for pctrti2

COMPLEX*16 for pztrti2.

Pointer into the local memory to an array, size(*l1d_a*, LOCc(*ja+n-1*)).

On entry, this array contains the local pieces of the triangular matrix sub(A).

If *uplo* = 'U', the leading n -by- n upper triangular part of the matrix sub(A) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(A) is not referenced.

If $uplo = 'L'$, the leading n -by- n lower triangular part of the matrix sub(A) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(A) is not referenced. If $diag = 'U'$, the diagonal elements of sub(A) are not referenced either and are assumed to be 1.

ia, ja (global) INTEGER.

The row and column indices in the global array A indicating the first row and the first column of the sub(A), respectively.

$desca$ (global and local) INTEGER array, size ($dlen_$). The array descriptor for the distributed matrix A .

Output Parameters

a On exit, the (triangular) inverse of the original matrix, in the same storage format.

$info$ INTEGER.

= 0: successful exit
 < 0: if the i -th argument is an array and the j -entry had an illegal value,
 then $info = - (i*100)$,
 if the i -th argument is a scalar and had an illegal value,
 then $info = -i$.

?lamsh

Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.

Syntax

Fortran:

```
call slamsh(s, lds, nbudge, jblk, h, ldh, n, ulp)
call dlamsh(s, lds, nbudge, jblk, h, ldh, n, ulp)
```

C:

```
void slamsh (float *s , MKL_INT *lds , MKL_INT *nbudge , MKL_INT *jblk , float *h ,
MKL_INT *ldh , MKL_INT *n , float *ulp );
void dlamsh (double *s , MKL_INT *lds , MKL_INT *nbudge , MKL_INT *jblk , double *h ,
MKL_INT *ldh , MKL_INT *n , double *ulp );
```

Include Files

- C: mkl_scalapack.h

Description

The ?lamsh routine sends multiple shifts through a small (single node) matrix to see how small consecutive subdiagonal elements are modified by subsequent shifts in an effort to maximize the number of bulges that can be sent through. The subroutine should only be called when there are multiple shifts/bulges ($nbulge > 1$) and the first shift is starting in the middle of an unreduced Hessenberg matrix because of two or more small consecutive subdiagonal elements.

Input Parameters

<i>s</i>	(local) INTEGER. REAL for slamsh DOUBLE PRECISION for dlamsh Array, size (<i>lds</i> ,*).
	On entry, the matrix of shifts. Only the 2×2 diagonal of <i>s</i> is referenced. It is assumed that <i>s</i> has <i>jblk</i> double shifts (size 2).
<i>lds</i>	(local) INTEGER. On entry, the leading dimension of <i>S</i> ; unchanged on exit. $1 < nbulge \leq jblk \leq lds/2$.
<i>nbulge</i>	(local) INTEGER. On entry, the number of bulges to send through <i>h</i> (>1). <i>nbulge</i> should be less than the maximum determined (<i>jblk</i>). $1 < nbulge \leq jblk \leq lds/2$.
<i>jblk</i>	(local) INTEGER. On entry, the leading dimension of <i>S</i> ; unchanged on exit.
<i>h</i>	(local) INTEGER. REAL for slamsh DOUBLE PRECISION for dlamsh Array, size <i>lds</i> by <i>n</i> . On entry, the local matrix to apply the shifts on. <i>h</i> should be aligned so that the starting row is 2.
<i>ldh</i>	(local) INTEGER. On entry, the leading dimension of <i>H</i> ; unchanged on exit.
<i>n</i>	(local) INTEGER. On entry, the size of <i>H</i> . If all the bulges are expected to go through, <i>n</i> should be at least $4nbulge+2$. Otherwise, <i>nbulge</i> may be reduced by this routine.
<i>ulp</i>	(local) REAL for slamsh DOUBLE PRECISION for dlamsh

On entry, machine precision. Unchanged on exit.

Output Parameters

<i>s</i>	On exit, the data is rearranged in the best order for applying.
<i>nbulge</i>	On exit, the maximum number of bulges that can be sent through.
<i>h</i>	On exit, the data is destroyed.

?laqr6

Performs a single small-bulge multi-shift QR sweep collecting the transformations.

Syntax

Fortran:

```
call slaqr6( job, wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh, iloz,
ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
call dlaqr6( job, wantt, wantz, kacc22, n, ktop, kbot, nshfts, sr, si, h, ldh, iloz,
ihiz, z, ldz, v, ldv, u, ldu, nv, wv, ldwv, nh, wh, ldwh )
```

Include Files

- C: mkl_scalapack.h

Description

This auxiliary subroutine called by [p?laqr5](#) performs a single small-bulge multi-shift QR sweep, moving the chain of bulges from top to bottom in the submatrix $h(ktop:kbot, ktop:kbot)$, collecting the transformations in the matrix V or accumulating the transformations in the matrix z (see below).

This is a modified version of [?laqr5](#) from LAPACK 3.1.

Input Parameters

<i>job</i>	CHARACTER scalar
	Set the kind of job to do in ?laqr6, as follows:
<i>job</i> = 'I':	Introduce and chase bulges in submatrix
<i>job</i> = 'C':	Chase bulges from top to bottom of submatrix
<i>job</i> = 'O':	Chase bulges off submatrix
<i>wantt</i>	LOGICAL scalar
	<i>wantt</i> = .TRUE. if the quasi-triangular Schur factor is being computed. <i>wantt</i> is set to .FALSE. otherwise.
<i>wantz</i>	LOGICAL scalar
	<i>wantz</i> = .TRUE. if the orthogonal Schur factor is being computed. <i>wantz</i> is set to .FALSE. otherwise.
<i>kacc22</i>	INTEGER with value 0, 1, or 2.

Specifies the computation mode of far-from-diagonal orthogonal updates.

= 0: ?laqr6 does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries.

= 1: ?laqr6 accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.

= 2: ?laqr6 accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.

n

INTEGER scalar

n is the order of the Hessenberg matrix *h* upon which this subroutine operates.

ktop, kbot

INTEGER scalar

These are the first and last rows and columns of an isolated diagonal block upon which the QR sweep is to be applied. It is assumed without a check that either *ktop* = 1 or *h(ktop,ktop-1)* = 0 and either *kbot* = *n* or *h(kbot+1,kbot)* = 0.

nshfts

INTEGER scalar

nshfts gives the number of simultaneous shifts. *nshfts* must be positive and even.

sr, si

REAL for slaqr6

DOUBLE PRECISION for dlaqr6

Array of size (*nshfts*)

sr contains the real parts and *si* contains the imaginary parts of the *nshfts* shifts of origin that define the multi-shift QR sweep.

h

REAL for slaqr6

DOUBLE PRECISION for dlaqr6

Array of size (*ldh,n*)

On input *h* contains a Hessenberg matrix.

ldh

INTEGER scalar

ldh is the leading dimension of *h* just as declared in the calling procedure. *ldh* $\geq \max(1,n)$.

iloz, ihiz

INTEGER scalar

Specify the rows of *z* to which transformations must be applied if *wantz* is .TRUE.. $1 \leq iloz \leq ihiz \leq n$

z

REAL for slaqr6

DOUBLE PRECISION for dlaqr6

Array of size (*ldz,ktop*)

If *wantz* = .TRUE., then the QR sweep orthogonal similarity transformation is accumulated into *z(iloz:ihiz,kbot:ktop)* from the right.

If $wantz = .FALSE.$, then z is unreferenced.

ldz

INTEGER scalar

ldz is the leading dimension of z just as declared in the calling procedure.
 $ldz \geq n$.

v

REAL for `slaqr6`

DOUBLE PRECISION for `dlaqr6`

(workspace) array of size ($ldv, nshfts/2$)

ldv

INTEGER scalar

ldv is the leading dimension of v as declared in the calling procedure.
 $ldv \geq 3$.

u

REAL for `slaqr6`

DOUBLE PRECISION for `dlaqr6`

(workspace) array of size ($ldu, 3*nshfts-3$)

ldu

INTEGER scalar

ldu is the leading dimension of u just as declared in the in the calling subroutine. $ldu \geq 3*nshfts-3$.

nh

INTEGER scalar

nh is the number of columns in array wh available for workspace. $nh \geq 1$ is required for usage of this workspace, otherwise the updates of the far-from-diagonal elements will be updated without level 3 BLAS.

wh

REAL for `slaqr6`

DOUBLE PRECISION for `dlaqr6`

(workspace) array of size ($ldwh, nh$)

$ldwh$

INTEGER scalar

Leading dimension of wh just as declared in the calling procedure.
 $ldwh \geq 3*nshfts-3$.

nv

INTEGER scalar

nv is the number of rows in wv available for workspace. $nv \geq 1$ is required for usage of this workspace, otherwise the updates of the far-from-diagonal elements will be updated without level 3 BLAS.

wv

REAL for `slaqr6`

DOUBLE PRECISION for `dlaqr6`

(workspace) array of size ($ldwv, 3*nshfts-3$)

$ldwv$

INTEGER scalar

$ldwv$ is the leading dimension of wv as declared in the in the calling subroutine. $ldwv \geq nv$.

OUTPUT Parameters

<i>h</i>	A multi-shift QR sweep with shifts $sr(j)+i*si(j)$ is applied to the isolated diagonal block in rows and columns k_{top} through k_{bot} .
<i>z</i>	If <i>wantz</i> is .TRUE., then the QR sweep orthogonal/unitary similarity transformation is accumulated into $z(iloz:ihiz,kbot:ktop)$ from the right. If <i>wantz</i> is .FALSE., then <i>z</i> is unreferenced.

Application Notes

Notes

Based on contributions by Karen Braman and Ralph Byers, Department of Mathematics, University of Kansas, USA Robert Granat, Department of Computing Science and HPC2N, Umeå University, Sweden

?lar1va

Computes scaled eigenvector corresponding to given eigenvalue.

Syntax

Fortran:

```
call slar1va(n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcorr, work )
call dlar1va(n, b1, bn, lambda, d, l, ld, lld, pivmin, gaptol, z, wantnc, negcnt, ztz,
mingma, r, isuppz, nrminv, resid, rqcorr, work )
```

Include Files

- C: mkl_scalapack.h

Description

?slar1va computes the (scaled) *r*-th column of the inverse of the submatrix in rows *b1* through *bn* of the tridiagonal matrix $L D L^T - \lambda I$. When λ is close to an eigenvalue, the computed vector is an accurate eigenvector. Usually, *r* corresponds to the index where the eigenvector is largest in magnitude. The following steps accomplish this computation :

1. Stationary qd transform, $L D L^T - \lambda I = L_+ D_+ L_+^T$,
2. Progressive qd transform, $L D L^T - \lambda I = U_- D_- U_-^T$,
3. Computation of the diagonal elements of the inverse of $L D L^T - \lambda I$ by combining the above transforms, and choosing *r* as the index where the diagonal of the inverse is (one of the) largest in magnitude.
4. Computation of the (scaled) *r*-th column of the inverse using the twisted factorization obtained by combining the top part of the stationary and the bottom part of the progressive transform.

Input Parameters

<i>n</i>	INTEGER
	The order of the matrix $L D L^T$.

<i>b1</i>	INTEGER
	First index of the submatrix of $L D L^T$.

<i>bn</i>	INTEGER
	Last index of the submatrix of $L D L^T$.
<i>lambda</i>	REAL for slarlva DOUBLE PRECISION for dlarlva
	The shift λ . In order to compute an accurate eigenvector, <i>lambda</i> should be a good approximation to an eigenvalue of $L D L^T$.
<i>l</i>	REAL for slarlva DOUBLE PRECISION for dlarlva Array, dimension ($n-1$)
	The $(n-1)$ subdiagonal elements of the unit bidiagonal matrix <i>l</i> , in elements 1 to $n-1$.
<i>d</i>	REAL for slarlva DOUBLE PRECISION for dlarlva Array, dimension (n)
	The n diagonal elements of the diagonal matrix <i>d</i> .
<i>ld</i>	REAL for slarlva DOUBLE PRECISION for dlarlva Array, dimension ($n-1$)
	The $n-1$ elements $l(i)*d(i)$.
<i>lld</i>	REAL for slarlva DOUBLE PRECISION for dlarlva Array, dimension ($n-1$)
	The $n-1$ elements $l(i)*l(i)*d(i)$.
<i>pivmin</i>	REAL for slarlva DOUBLE PRECISION for dlarlva
	The minimum pivot in the Sturm sequence.
<i>gaptol</i>	REAL for slarlva DOUBLE PRECISION for dlarlva
	Tolerance that indicates when eigenvector entries are negligible with respect to their contribution to the residual.
<i>z</i>	REAL for slarlva DOUBLE PRECISION for dlarlva Array, dimension (n)
	On input, all entries of <i>z</i> must be set to 0.
<i>wantnc</i>	LOGICAL

Specifies whether *negcnt* has to be computed.

r

INTEGER

The twist index for the twisted factorization used to compute *z*.

On input, $0 \leq r \leq n$. If *r* is input as 0, *r* is set to the index where $(L D L^T - \sigma I)^{-1}$ is largest in magnitude. If $1 \leq r \leq n$, *r* is unchanged.

Ideally, *r* designates the position of the maximum entry in the eigenvector.

work

REAL for slarlva

DOUBLE PRECISION for dlarlva

(workspace) array, dimension ($4 * n$)

OUTPUT Parameters

z

On output, *z* contains the (scaled) *r*-th column of the inverse. The scaling is such that *z(r)* equals 1.

negcnt

INTEGER

If *wantnc* is .TRUE., then *negcnt* = the number of pivots < *pivmin* in the matrix factorization $L D L^T$, and *negcnt* = -1 otherwise.

ztz

REAL for slarlva

DOUBLE PRECISION for dlarlva

The square of the 2-norm of *z*.

mingma

REAL for slarlva

DOUBLE PRECISION for dlarlva

The reciprocal of the largest (in magnitude) diagonal element of the inverse of $L D L^T - \sigma I$.

r

On output, *r* contains the twist index used to compute *z*.

isuppz

INTEGER array, dimension (2)

The support of the vector in *z*, i.e., the vector *z* is nonzero only in elements *isuppz(1)* through *isuppz(2)*.

nrminv

REAL for slarlva

DOUBLE PRECISION for dlarlva

nrminv = $1/\text{SQRT}(ztz)$

resid

REAL for slarlva

DOUBLE PRECISION for dlarlva

The residual of the FP vector.

resid = $\text{ABS}(mingma)/\text{SQRT}(ztz)$

rqcorr

REAL for slarlva

DOUBLE PRECISION for dlarlva
 The Rayleigh Quotient correction to *lambda*.

?laref

Applies Householder reflectors to matrices on either their rows or columns.

Syntax

Fortran:

```
call slaref(type, a, lda, wantz, z, ldz, block, irow1, icoll, istart, istop, itmp1,
itmp2, liloz, lihiz, vecs, v2, v3, t1, t2, t3)

call dlaref(type, a, lda, wantz, z, ldz, block, irow1, icoll, istart, istop, itmp1,
itmp2, liloz, lihiz, vecs, v2, v3, t1, t2, t3)
```

C:

```
void slaref (char *type , float *a , MKL_INT *lda , MKL_INT *wantz , float *z ,
MKL_INT *ldz , MKL_INT *block , MKL_INT *irow1 , MKL_INT *icoll , MKL_INT *istart ,
MKL_INT *istop , MKL_INT *itmp1 , MKL_INT *itmp2 , MKL_INT *liloz , MKL_INT *lihiz ,
float *vecs , float *v2 , float *v3 , float *t1 , float *t2 , float *t3 );

void dlaref (char *type , double *a , MKL_INT *lda , MKL_INT *wantz , double *z ,
MKL_INT *ldz , MKL_INT *block , MKL_INT *irow1 , MKL_INT *icoll , MKL_INT *istart ,
MKL_INT *istop , MKL_INT *itmp1 , MKL_INT *itmp2 , MKL_INT *liloz , MKL_INT *lihiz ,
double *vecs , double *v2 , double *v3 , double *t1 , double *t2 , double *t3 );
```

Include Files

- C: mkl_scalapack.h

Description

The ?laref routine applies one or several Householder reflectors of size 3 to one or two matrices (if column is specified) on either their rows or columns.

Input Parameters

type (global) CHARACTER*1.
 If *type* = 'R', apply reflectors to the rows of the matrix (apply from left). Otherwise, apply reflectors to the columns of the matrix. Unchanged on exit.

a (global) REAL for slaref
 DOUBLE PRECISION for dlaref
 Array, size (*lda*, *).
 On entry, the matrix to receive the reflections.

lda (local) INTEGER.
 On entry, the leading dimension of *A*; unchanged on exit.

wantz (global) LOGICAL.

If *wantz* = .TRUE., apply any column reflections to *Z* as well.

If *wantz* = .FALSE., do no additional work on *Z*.

z (global) REAL for slaref

DOUBLE PRECISION for dlaref

Array, size (*ldz*, *).

On entry, the second matrix to receive column reflections.

ldz (local) INTEGER.

On entry, the leading dimension of *Z*; unchanged on exit.

block (global) LOGICAL.

= .TRUE. : apply several reflectors at once and read their data from the *vecs* array;

= .FALSE. : apply the single reflector given by *v2*, *v3*, *t1*, *t2*, and *t3*.

irow1 (local) INTEGER.

On entry, the local row element of the matrix *A*.

icoll (local) INTEGER.

On entry, the local column element of the matrix *A*.

istart (global) INTEGER.

Specifies the "number" of the first reflector.

istart is used as an index into *vecs* if *block* is set. *istart* is ignored if *block* is .FALSE. .

istop (global) INTEGER.

Specifies the "number" of the last reflector.

istop is used as an index into *vecs* if *block* is set. *istop* is ignored if *block* is .FALSE. .

itmp1 (local) INTEGER.

Starting range into *A*. For rows, this is the local first column. For columns, this is the local first row.

itmp2 (local) INTEGER.

Ending range into *A*. For rows, this is the local last column. For columns, this is the local last row.

liloz, *lihiz* (local). INTEGER.

Serve the same purpose as *itmp1*, *itmp2* but for *Z* when *wantz* is set.

vecs (global)

REAL for slaref

DOUBLE PRECISION for dlaref.

Array of size $3 * n$ (matrix size). This array holds the size 3 reflectors one after another and is only accessed when *block* is .TRUE..

```
v2,v3,t1,t2,t3          (global) INTEGER.  
REAL for slaref  
DOUBLE PRECISION for dlaref.
```

These parameters hold information on a single size 3 Householder reflector and are read when *block* is .FALSE., and overwritten when *block* is .TRUE..

Output Parameters

<i>a</i>	On exit, the updated matrix.
<i>z</i>	Changed only if <i>wantz</i> is set. If <i>wantz</i> is .FALSE., <i>z</i> is not referenced.
<i>irow1</i>	Undefined.
<i>icoll</i>	Undefined.
<i>v2,v3,t1,t2,t3</i>	These parameters are read when <i>block</i> is .FALSE., and overwritten when <i>block</i> is .TRUE..

?larrb2

Provides limited bisection to locate eigenvalues for more accuracy.

Syntax

Fortran:

```
call slarrb2( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work,  
iwork, pivmin, lgpvmn, lgspdm, twist, info )  
call dlarrb2( n, d, lld, ifirst, ilast, rtol1, rtol2, offset, w, wgap, werr, work,  
iwork, pivmin, lgpvmn, lgspdm, twist, info )
```

Include Files

- C: mkl_scalapack.h

Description

Given the relatively robust representation (RRR) $L D L^T$, ?larrb2 does "limited" bisection to refine the eigenvalues of $L D L^T$, $w(ifirst - offset)$ through $w(ilast - offset)$, to more accuracy. Initial guesses for these eigenvalues are input in *w*, the corresponding estimate of the error in these guesses and their gaps are input in *werr* and *wgap*, respectively. During bisection, intervals $[left, right]$ are maintained by storing their mid-points and semi-widths in the arrays *w* and *werr* respectively.

NOTE

There are very few minor differences between `?larrb` from LAPACK and this current subroutine `?larrb2`. The most important reason for creating this nearly identical copy is profiling: in the ScaLAPACK MRRR algorithm, eigenvalue computation using `?larrb2` is used for refinement in the construction of the representation tree, as opposed to the initial computation of the eigenvalues for the root RRR which uses `?larrb`. When profiling, this allows an easy quantification of refinement work vs. computing eigenvalues of the root.

Input Parameters

<i>n</i>	INTEGER	
		The order of the matrix.
<i>d</i>	REAL for <code>slarrb2</code> DOUBLE PRECISION for <code>dlarrb2</code>	
		Array, dimension (<i>n</i>)
		The <i>n</i> diagonal elements of the diagonal matrix <i>D</i> .
<i>lld</i>	REAL for <code>slarrb2</code> DOUBLE PRECISION for <code>dlarrb2</code>	
		Array, dimension (<i>n</i> -1)
		The (<i>n</i> -1) elements $l_i^* l_i^* d(i)$.
<i>ifirst</i>	INTEGER	
		The index of the first eigenvalue to be computed.
<i>ilast</i>	INTEGER	
		The index of the last eigenvalue to be computed.
<i>rtol1, rtol2</i>	REAL for <code>slarrb2</code> DOUBLE PRECISION for <code>dlarrb2</code>	
		Tolerance for the convergence of the bisection intervals.
		An interval $[left, right]$ has converged if $right - left < \max(rtolt * gap, rtolt * \max(left , right))$ where <i>gap</i> is the (estimated) distance to the nearest eigenvalue.
<i>offset</i>	INTEGER	
		Offset for the arrays <i>w</i> , <i>wgap</i> and <i>werr</i> , i.e., the <i>ifirst</i> - <i>offset</i> through <i>ilast</i> - <i>offset</i> elements of these arrays are to be used.
<i>w</i>	REAL for <code>slarrb2</code> DOUBLE PRECISION for <code>dlarrb2</code>	
		Array, dimension (<i>n</i>)
		On input, <i>w</i> (<i>ifirst</i> - <i>offset</i>) through <i>w</i> (<i>ilast</i> - <i>offset</i>) are estimates of the eigenvalues of $L D L^T$ indexed <i>ifirst</i> through <i>ilast</i> .

<i>wgap</i>	REAL for slarrb2 DOUBLE PRECISION for dlarrb2 Array, dimension ($n-1$) On input, the (estimated) gaps between consecutive eigenvalues of $L D L^T$, i.e., $wgap(I - \text{offset})$ is the gap between eigenvalues I and $I + 1$. Note that if $\text{ifirst} = \text{ilast}$ then $wgap(\text{ifirst} - \text{offset})$ must be set to zero.
<i>werr</i>	REAL for slarrb2 DOUBLE PRECISION for dlarrb2 Array, dimension (n) On input, $werr(\text{ifirst} - \text{offset})$ through $werr(\text{ilast} - \text{offset})$ are the errors in the estimates of the corresponding elements in w .
<i>work</i>	REAL for slarrb2 DOUBLE PRECISION for dlarrb2 (workspace) array, dimension ($4*n$) Workspace.
<i>iwork</i>	(workspace) INTEGER array, dimension ($2*n$) Workspace.
<i>pivmin</i>	REAL for slarrb2 DOUBLE PRECISION for dlarrb2 The minimum pivot in the sturm sequence.
<i>lgpvmn</i>	REAL for slarrb2 DOUBLE PRECISION for dlarrb2 Logarithm of $pivmin$, precomputed.
<i>lgspdm</i>	REAL for slarrb2 DOUBLE PRECISION for dlarrb2 Logarithm of the spectral diameter, precomputed.
<i>twist</i>	INTEGER The twist index for the twisted factorization that is used for the <i>negcount</i> . $twist = n$: Compute <i>negcount</i> from $L D L^T - \lambda I = L_+ D_+ L_+^T$ $twist = 1$: Compute <i>negcount</i> from $L D L^T - \lambda I = U_- D_- U_-^T$ $twist = r$, $1 < r < n$: Compute <i>negcount</i> from $L D L^T - \lambda I = N_r \Delta_r N_r^T$

OUTPUT Parameters

w On output, the eigenvalue estimates in w are refined.

wgap On output, the eigenvalue gaps in $wgap$ are refined.

werr On output, the errors in *werr* are refined.

info INTEGER

Error flag.

?larrd2

Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.

Syntax

Fortran:

```
call slarrd2( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplitt,
isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, dol, dou, info )
call dlarrd2( range, order, n, vl, vu, il, iu, gers, reltol, d, e, e2, pivmin, nsplitt,
isplit, m, w, werr, wl, wu, iblock, indexw, work, iwork, dol, dou, info )
```

Include Files

- C: mkl_scalapack.h

Description

?larrd2 computes the eigenvalues of a symmetric tridiagonal matrix T to limited initial accuracy. This is an auxiliary code to be called from ?larre2a.

?larrd2 has been created using the LAPACK code ?larrd which itself stems from ?stebz. The motivation for creating ?larrd2 is efficiency: When computing eigenvalues in parallel and the input tridiagonal matrix splits into blocks, ?larrd2 can skip over blocks which contain none of the eigenvalues from DOL to DOU for which the processor responsible. In extreme cases (such as large matrices consisting of many blocks of small size like 2x2), the gain can be substantial.

Input Parameters

<i>range</i>	CHARACTER
= 'A': ("All")	all eigenvalues will be found.
= 'V': ("Value")	all eigenvalues in the half-open interval (<i>vl</i> , <i>vu</i>] will be found.
= 'I': ("Index")	the <i>il</i> -th through <i>iu</i> -th eigenvalues (of the entire matrix) will be found.
<i>order</i>	CHARACTER
= 'B': ("By Block")	the eigenvalues will be grouped by split-off block (see <i>iblock</i> , <i>isplit</i>) and ordered from smallest to largest within the block.
= 'E': ("Entire matrix")	the eigenvalues for the entire matrix will be ordered from smallest to largest.
<i>n</i>	INTEGER
	The order of the tridiagonal matrix T . $n \geq 0$.
<i>vl</i> , <i>vu</i>	REAL for slarrd2

DOUBLE PRECISION for `dlarrd2`

If `range='V'`, the lower and upper bounds of the interval to be searched for eigenvalues. Eigenvalues less than or equal to `vl`, or greater than `vu`, will not be returned. $vl < vu$.

Not referenced if `range = 'A'` or `'I'`.

`il, iu`

INTEGER

If `range='I'`, the indices (in ascending order) of the smallest and largest eigenvalues to be returned.

$1 \leq il \leq iu \leq n$, if $n > 0$; $il = 1$ and $iu = 0$ if $n = 0$.

Not referenced if `range = 'A'` or `'V'`.

`gers`

REAL for `slarrd2`

DOUBLE PRECISION for `dlarrd2`

Array, dimension (2^*n)

The n Gershgorin intervals (the i -th Gershgorin interval is $(gers(2*i-1), gers(2*i))$).

`reltol`

REAL for `slarrd2`

DOUBLE PRECISION for `dlarrd2`

The minimum relative width of an interval. When an interval is narrower than `reltol` times the larger (in magnitude) endpoint, then it is considered to be sufficiently small, i.e., converged. Note: this should always be at least radix*machine epsilon.

`d`

REAL for `slarrd2`

DOUBLE PRECISION for `dlarrd2`

Array, dimension (n)

The n diagonal elements of the tridiagonal matrix T .

`e`

REAL for `slarrd2`

DOUBLE PRECISION for `dlarrd2`

Array, dimension ($n-1$)

The $(n-1)$ off-diagonal elements of the tridiagonal matrix T .

`e2`

REAL for `slarrd2`

DOUBLE PRECISION for `dlarrd2`

Array, dimension ($n-1$)

The $(n-1)$ squared off-diagonal elements of the tridiagonal matrix T .

`pivmin`

REAL for `slarrd2`

DOUBLE PRECISION for `dlarrd2`

The minimum pivot allowed in the sturm sequence for T .

`nsplit`

INTEGER

The number of diagonal blocks in the matrix T .

$1 \leq nsplit \leq n$.

isplit

INTEGER Array, dimension (n)

The splitting points, at which T breaks up into submatrices.

The first submatrix consists of rows/columns 1 to $isplit(1)$, the second of rows/columns $isplit(1)+1$ through $isplit(2)$, etc., and the $nsplit$ -th consists of rows/columns $isplit(nsplit-1)+1$ through $isplit(nsplit)=n$.

(Only the first $nsplit$ elements will actually be used, but since the user cannot know a priori what value $nsplit$ will have, n words must be reserved for *isplit*.)

work

REAL for slarrd2

DOUBLE PRECISION for dlarrd2

(workspace) array, dimension ($4*n$)

iwork

(workspace) INTEGER array, dimension ($3*n$)

dol, dou

INTEGER

Specifying an index range $dol:dou$ allows the user to work on only a selected part of the representation tree.

Otherwise, the setting $dol=1, dou=n$ should be applied.

Note that *dol* and *dou* refer to the order in which the eigenvalues are stored in *W*.

OUTPUT Parameters

m

INTEGER

The actual number of eigenvalues found. $0 \leq m \leq n$.

(See also the description of *info=2,3*.)

w

REAL for slarrd2

DOUBLE PRECISION for dlarrd2

Array, dimension (n)

On exit, the first *m* elements of *w* will contain the eigenvalue approximations. ?larrd2 computes an interval $I_j = (a_j, b_j]$ that includes eigenvalue *j*. The eigenvalue approximation is given as the interval midpoint $w(j) = (a_j + b_j)/2$. The corresponding error is bounded by $werr(j) = \text{abs}(a_j - b_j)/2$.

werr

REAL for slarrd2

DOUBLE PRECISION for dlarrd2

Array, dimension (n)

The error bound on the corresponding eigenvalue approximation in *w*.

<i>wl, wu</i>	REAL for slarrd2 DOUBLE PRECISION for dlarrd2
	The interval (<i>wl, wu</i>] contains all the wanted eigenvalues.
	If <i>range</i> ='V', then <i>wl</i> = <i>vl</i> and <i>wu</i> = <i>vu</i> .
	If <i>range</i> ='A', then <i>wl</i> and <i>wu</i> are the global Gershgorin bounds on the spectrum.
	If <i>range</i> ='I', then <i>wl</i> and <i>wu</i> are computed by SLAEBZ from the index range specified.
<i>iblock</i>	INTEGER Array, dimension (<i>n</i>) At each row/column <i>j</i> where <i>e(j)</i> is zero or small, the matrix <i>T</i> is considered to split into a block diagonal matrix. On exit, if <i>info</i> = 0, <i>iblock(i)</i> specifies to which block (from 1 to the number of blocks) the eigenvalue <i>w(i)</i> belongs. (?larrd2 may use the remaining <i>n-m</i> elements as workspace.)
<i>indexw</i>	INTEGER Array, dimension (<i>n</i>) The indices of the eigenvalues within each block (submatrix); for example, <i>indexw(i)=j</i> and <i>iblock(i)=k</i> imply that the <i>i</i> -th eigenvalue <i>w(i)</i> is the <i>j</i> -th eigenvalue in block <i>k</i> .
<i>info</i>	INTEGER = 0: successful exit < 0: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value > 0: some or all of the eigenvalues failed to converge or were not computed: <ul style="list-style-type: none">• =1 or 3: Bisection failed to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.• =2 or 3: <i>range</i>='I' only: Not all of the eigenvalues <i>il:iu</i> were found.• = 4: <i>range</i>='I', and the Gershgorin interval initially used was too small. No eigenvalues were computed.

?larre2

Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.

Syntax

Fortran:

```
call slarre2( range, n, vl, vu, il, iu, d, e, e2, rtoll, rtol2, spltol, nsplit, isplit, m, dol, dou, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
call dlarre2( range, n, vl, vu, il, iu, d, e, e2, rtoll, rtol2, spltol, nsplit, isplit, m, dol, dou, w, werr, wgap, iblock, indexw, gers, pivmin, work, iwork, info )
```

Include Files

- C: mkl_scalapack.h

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T , ?larre2 sets, via ?larra, "small" off-diagonal elements to zero. For each block T_i , it finds

- a suitable shift at one end of the block's spectrum,
- the root RRR, $T_i - \sigma_i I = L_i D_i L_i^T$, and
- eigenvalues of each $L_i D_i L_i^T$.

The representations and eigenvalues found are then returned to ?stegr2 to compute the eigenvectors T .

?larre2 is more suitable for parallel computation than the original LAPACK code for computing the root RRR and its eigenvalues. When computing eigenvalues in parallel and the input tridiagonal matrix splits into blocks, ?larre2 can skip over blocks which contain none of the eigenvalues from *d0l* to *dou* for which the processor responsible. In extreme cases (such as large matrices consisting of many blocks of small size, e.g. 2x2), the gain can be substantial.

Input Parameters

<i>range</i>	CHARACTER
	= 'A': ("All") all eigenvalues will be found.
	= 'V': ("Value") all eigenvalues in the half-open interval (<i>vl</i> , <i>vu</i>] will be found.
	= 'I': ("Index") the <i>il</i> -th through <i>iu</i> -th eigenvalues (of the entire matrix) will be found.
<i>n</i>	INTEGER
	The order of the matrix. <i>n</i> > 0.
<i>vl</i> , <i>vu</i>	REAL for slarre2 DOUBLE PRECISION for dlarre2 If <i>range</i> ='V', the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to <i>vl</i> , or greater than <i>vu</i> , will not be returned. <i>vl</i> < <i>vu</i> .
<i>il</i> , <i>iu</i>	INTEGER If <i>range</i> ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq il \leq iu \leq n$.
<i>d</i>	REAL for slarre2 DOUBLE PRECISION for dlarre2 Array, dimension (<i>n</i>) The <i>n</i> diagonal elements of the tridiagonal matrix T .
<i>e</i>	REAL for slarre2 DOUBLE PRECISION for dlarre2

Array, dimension (n)

The first ($n-1$) entries contain the subdiagonal elements of the tridiagonal matrix T ; $e(n)$ need not be set.

$e2$

REAL for slarre2

DOUBLE PRECISION for dlarre2

Array, dimension (n)

The first ($n-1$) entries contain the squares of the subdiagonal elements of the tridiagonal matrix T ; $e2(n)$ need not be set.

$rto11, rto12$

REAL for slarre2

DOUBLE PRECISION for dlarre2

Parameters for bisection.

An interval $[left, right]$ has converged if $right-left < \max(rto11*gap, rto12*\max(|left|, |right|))$

$spltol$

REAL for slarre2

DOUBLE PRECISION for dlarre2

The threshold for splitting.

dol, dou

INTEGER

Specifying an index range $dol:dou$ allows the user to work on only a selected part of the representation tree. Otherwise, the setting $dol=1$, $dou=n$ should be applied.

Note that dol and dou refer to the order in which the eigenvalues are stored in w .

$work$

REAL for slarre2

DOUBLE PRECISION for dlarre2

Workspace array, dimension ($6*n$)

$iwork$

INTEGER workspace array, dimension ($5*n$)

OUTPUT Parameters

vl, vu

If $range='I'$ or $'A'$, ?larre2 contains bounds on the desired part of the spectrum.

d

The n diagonal elements of the diagonal matrices D_i .

e

e contains the subdiagonal elements of the unit bidiagonal matrices L_i . The entries $e(isplit(i))$, $1 \leq i \leq nsplit$, contain the base points σ_i on output.

$e2$

The entries $e2(isplit(i))$, $1 \leq i \leq nsplit$, are set to zero.

$nsplit$

INTEGER

The number of blocks T splits into. $1 \leq nsplit \leq n$.

<i>isplit</i>	INTEGER Array, dimension (<i>n</i>) The splitting points, at which <i>T</i> breaks up into blocks. The first block consists of rows/columns 1 to <i>isplit</i> (1), the second of rows/columns <i>isplit</i> (1)+1 through <i>isplit</i> (2), etc., and the <i>nsplit</i> -th consists of rows/columns <i>isplit</i> (<i>nsplit</i> -1)+1 through <i>isplit</i> (<i>nsplit</i>)= <i>n</i> .
<i>m</i>	INTEGER The total number of eigenvalues (of all $L_i D_i L_i^T$) found.
<i>w</i>	REAL for slarre2 DOUBLE PRECISION for dlarre2 Array, dimension (<i>n</i>) The first <i>m</i> elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i D_i L_i^T$, are sorted in ascending order (?larre2 may use the remaining <i>n-m</i> elements as workspace). Note that immediately after exiting this routine, only the eigenvalues from position <i>dol:dou</i> in <i>w</i> might rely on this processor when the eigenvalue computation is done in parallel.
<i>werr</i>	REAL for slarre2 DOUBLE PRECISION for dlarre2 Array, dimension (<i>n</i>) The error bound on the corresponding eigenvalue in <i>w</i> . Note that immediately after exiting this routine, only the uncertainties from position <i>dol:dou</i> in <i>werr</i> might rely on this processor when the eigenvalue computation is done in parallel.
<i>wgap</i>	REAL for slarre2 DOUBLE PRECISION for dlarre2 Array, dimension (<i>n</i>) The separation from the right neighbor eigenvalue in <i>w</i> . The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree. Exception: at the right end of a block we store the left gap Note that immediately after exiting this routine, only the gaps from position <i>dol:dou</i> in <i>wgap</i> might rely on this processor when the eigenvalue computation is done in parallel.
<i>iblock</i>	INTEGER Array, dimension (<i>n</i>) The indices of the blocks (submatrices) associated with the corresponding eigenvalues in <i>w</i> ; <i>iblock</i> (<i>i</i>)=1 if eigenvalue <i>w</i> (<i>i</i>) belongs to the first block from the top, <i>iblock</i> (<i>i</i>)=2 if <i>w</i> (<i>i</i>) belongs to the second block, and so on.
<i>indexw</i>	INTEGER Array, dimension (<i>n</i>)

The indices of the eigenvalues within each block (submatrix); for example, $indexw(i) = 10$ and $iblock(i)=2$ imply that the i -th eigenvalue $w(i)$ is the 10th eigenvalue in block 2.

gers REAL for slarre2
 DOUBLE PRECISION for dlarre2
 Array, dimension (2^*n)
 The n Gerschgorin intervals (the i -th Gerschgorin interval is $(gers(2*i-1), gers(2*i))$).

pivmin REAL for slarre2
 DOUBLE PRECISION for dlarre2
 The minimum pivot in the sturm sequence for T .

info INTEGER
 = 0: successful exit
 > 0: A problem occurred in ?larre2.
 < 0: One of the called subroutines signaled an internal problem.
 Needs inspection of the corresponding parameter *info* for further information.
 =-1: Problem in ?larrd .
 =-2: Not enough internal iterations to find the base representation.
 =-3: Problem in ?larrb when computing the refined root representation for ?lasq2.
 =-4: Problem in ?larrb when performing bisection on the desired part of the spectrum.
 =-5: Problem in ?lasq2
 =-6: Problem in ?lasq2

?larre2a

Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.

Syntax

Fortran:

```
call slarre2a( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit,
isplit, m, dol, dou, needil, neediu, w, werr, wgap, iblock, indexw, gers, sdiam,
pivmin, work, iwork, minrgp, info )  

call dlarre2a( range, n, vl, vu, il, iu, d, e, e2, rtol1, rtol2, spltol, nsplit,
isplit, m, dol, dou, needil, neediu, w, werr, wgap, iblock, indexw, gers, sdiam,
pivmin, work, iwork, minrgp, info )
```

Include Files

- C: mkl_scalapack.h

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix T , ?larre2a sets any "small" off-diagonal elements to zero, and for each unreduced block T_i , it finds

- a suitable shift at one end of the block's spectrum,
- the base representation, $T_i - \sigma_i I = L_i D_i L_i^T$, and
- eigenvalues of each $L_i D_i L_i^T$.

NOTE

The algorithm obtains a crude picture of all the wanted eigenvalues (as selected by *range*). However, to reduce work and improve scalability, only the eigenvalues *d0l* to *d0u* are refined. Furthermore, if the matrix splits into blocks, RRRs for blocks that do not contain eigenvalues from *d0l* to *d0u* are skipped. The DQDS algorithm (subroutine ?1asq2) is not used, unlike in the sequential case. Instead, eigenvalues are computed in parallel to some figures using bisection.

Input Parameters

<i>range</i>	CHARACTER
	= 'A': ("All") all eigenvalues will be found.
	= 'V': ("Value") all eigenvalues in the half-open interval (<i>vl</i> , <i>vu</i>] will be found.
	= 'I': ("Index") the <i>i1</i> -th through <i>iu</i> -th eigenvalues (of the entire matrix) will be found.
<i>n</i>	INTEGER
	The order of the matrix. <i>n</i> > 0.
<i>vl</i> , <i>vu</i>	REAL for slarre2a DOUBLE PRECISION for dlarre2a If <i>range</i> ='V', the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to <i>vl</i> , or greater than <i>vu</i> , will not be returned. <i>vl</i> < <i>vu</i> . If <i>range</i> ='I' or ='A', ?larre2a computes bounds on the desired part of the spectrum.
<i>i1</i> , <i>iu</i>	INTEGER If <i>range</i> ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. $1 \leq i1 \leq iu \leq n$.
<i>d</i>	REAL for slarre2a DOUBLE PRECISION for dlarre2a Array, dimension (<i>n</i>) On entry, the <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i> .

e

REAL for slarre2a
DOUBLE PRECISION for dlarre2a
Array, dimension (n)
The first ($n-1$) entries contain the subdiagonal elements of the tridiagonal matrix T ; $e(n)$ need not be set.

e2

REAL for slarre2a
DOUBLE PRECISION for dlarre2a
Array, dimension (n)
The first ($n-1$) entries contain the squares of the subdiagonal elements of the tridiagonal matrix T ; $e2(n)$ need not be set.

rto11, rto12

REAL for slarre2a
DOUBLE PRECISION for dlarre2a
Parameters for bisection.
An interval $[left, right]$ has converged if $right - left < \max(rto11 * gap, rto12 * \max(|left|, |right|))$

spltol

REAL for slarre2a
DOUBLE PRECISION for dlarre2a
The threshold for splitting.

dol, dou

INTEGER
If the user wants to work on only a selected part of the representation tree, he can specify an index range $dol:dou$.
Otherwise, the setting $dol=1, dou=n$ should be applied.
Note that dol and dou refer to the order in which the eigenvalues are stored in w .

work

REAL for slarre2a
DOUBLE PRECISION for dlarre2a
Workspace array, dimension ($6*n$)

iwork

INTEGER workspace array, dimension ($5*n$)

minrgp

REAL for slarre2a
DOUBLE PRECISION for dlarre2a
The minimum relative gap threshold to decide whether an eigenvalue or a cluster boundary is reached.

OUTPUT Parameters

vl, vu

REAL for slarre2a
DOUBLE PRECISION for dlarre2a

If `range='V'`, the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to `vl`, or greater than `vu`, are not returned. $vl < vu$.

If `range='I'` or `range='A'`, ?larre2a computes bounds on the desired part of the spectrum.

d

The n diagonal elements of the diagonal matrices D_i .

e

e contains the subdiagonal elements of the unit bidiagonal matrices L_i . The entries $e(isplit(i))$, $1 \leq i \leq nsplit$, contain the base points σ_i on output.

e2

The entries $e2(isplit(i))$,
 $1 \leq i \leq nsplit$, have been set to zero.

nsplit

INTEGER

The number of blocks T splits into. $1 \leq nsplit \leq n$.

isplit

INTEGER array, dimension (n)

The splitting points, at which T breaks up into blocks. The first block consists of rows/columns 1 to $isplit(1)$, the second of rows/columns $isplit(1)+1$ through $isplit(2)$, etc., and the $nsplit$ -th consists of rows/columns $isplit(nsplit-1)+1$ through $isplit(nsplit)=n$.

m

INTEGER

The total number of eigenvalues (of all $L_i D_i L_i^T$) found.

needil, neediu

INTEGER

The indices of the leftmost and rightmost eigenvalues of the root node RRR which are needed to accurately compute the relevant part of the representation tree.

w

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

Array, dimension (n)

The first m elements contain the eigenvalues. The eigenvalues of each of the blocks, $L_i D_i L_i^T$, are sorted in ascending order (?larre2a may use the remaining $n-m$ elements as workspace).

Note that immediately after exiting this routine, only the eigenvalues from position `dol:dou` in *w* rely on this processor because the eigenvalue computation is done in parallel.

werr

REAL for slarre2a

DOUBLE PRECISION for dlarre2a

Array, dimension (n)

The error bound on the corresponding eigenvalue in *w*.

Note that immediately after exiting this routine, only the uncertainties from position `dol:dou` in *werr* are reliable on this processor because the eigenvalue computation is done in parallel.

wgap

REAL for slarre2a
 DOUBLE PRECISION for dlarre2a
 Array, dimension (n)
 The separation from the right neighbor eigenvalue in w . The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree.
 Exception: at the right end of a block we store the left gap
 Note that immediately after exiting this routine, only the gaps from position $dol:dou$ in *wgap* are reliable on this processor because the eigenvalue computation is done in parallel.

iblock

INTEGER Array, dimension (n)
 The indices of the blocks (submatrices) associated with the corresponding eigenvalues in w ; $iblock(i)=1$ if eigenvalue $w(i)$ belongs to the first block from the top, $iblock(i)=2$ if $w(i)$ belongs to the second block, etc.

indexw

INTEGER Array, dimension (n)
 The indices of the eigenvalues within each block (submatrix); for example, $indexw(i)=10$ and $iblock(i)=2$ imply that the i -th eigenvalue $w(i)$ is the 10-th eigenvalue in block 2.

gers

REAL for slarre2a
 DOUBLE PRECISION for dlarre2a
 Array, dimension (2^*n)
 The n Gerschgorin intervals (the i -th Gerschgorin interval is $(gers(2*i-1), gers(2*i))$).

pivmin

REAL for slarre2a
 DOUBLE PRECISION for dlarre2a
 The minimum pivot in the sturm sequence for T .

info

INTEGER
 = 0: successful exit
 > 0: A problem occurred in ?larre2a.
 < 0: One of the called subroutines signaled an internal problem. Needs inspection of the corresponding parameter *info* for further information.
 =-1: Problem in ?larrd2.
 =-2: Not enough internal iterations to find base representation.
 =-3: Problem in ?larrb2 when computing the refined root representation.
 =-4: Problem in ?larrb2 when performing bisection on the desired part of the spectrum.
 = -9 Problem: $m < dou-dol+1$, that is the code found fewer eigenvalues than it was supposed to.

?larrf2

Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.

Syntax**Fortran:**

```
call slarrf2( n, d, l, ld, clstrt, clegend, clmid1, clmid2, w, wgap, werr, trymid,
spdiam, clgap1, clgapr, pivmin, sigma, dplus, lplus, work, info )
```

```
call dlarrf2( n, d, l, ld, clstrt, clegend, clmid1, clmid2, w, wgap, werr, trymid,
spdiam, clgap1, clgapr, pivmin, sigma, dplus, lplus, work, info )
```

Include Files

- C: mkl_scalapack.h

Description

Given the initial representation $L D L^T$ and its cluster of close eigenvalues (in a relative measure), $w(c1strt)$, $w(c1strt+1)$, ..., $w(clegend)$, ?larrf2 finds a new relatively robust representation $L D L^T - \sigma I = L_+ D_+ L_+^T$ such that at least one of the eigenvalues of $L_+ D_+ L_+^T$ is relatively isolated.

This is an enhanced version of ?larrf that also tries shifts in the middle of the cluster, should there be a large gap, in order to break large clusters into at least two pieces.

Input Parameters

<i>n</i>	INTEGER	
		The order of the matrix (subblock, if the matrix was split).
<i>d</i>	REAL for slarrf2 DOUBLE PRECISION for dlarrf2	
		Array, dimension (<i>n</i>)
		The <i>n</i> diagonal elements of the diagonal matrix <i>D</i> .
<i>l</i>	REAL for slarrf2 DOUBLE PRECISION for dlarrf2	
		Array, dimension (<i>n</i> -1)
		The (<i>n</i> -1) subdiagonal elements of the unit bidiagonal matrix <i>L</i> .
<i>ld</i>	REAL for slarrf2 DOUBLE PRECISION for dlarrf2	
		Array, dimension (<i>n</i> -1)
		The (<i>n</i> -1) elements <i>l</i> (<i>i</i>)* <i>d</i> (<i>i</i>).
<i>clstrt</i>	INTEGER	
		The index of the first eigenvalue in the cluster.
<i>clegend</i>	INTEGER	

The index of the last eigenvalue in the cluster.

clmid1, clmid2

INTEGER

The index of a middle eigenvalue pair with large gap.

w

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Array, dimension $\geq (clend-clstrt+1)$

The eigenvalue approximations of $L D L^T$ in ascending order. *w*(*clstrt*) through *w*(*clend*) form the cluster of relatively close eigenvalues.

wgap

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Array, dimension $\geq (clend-clstrt+1)$

The separation from the right neighbor eigenvalue in *w*.

werr

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Array, dimension $\geq (clend-clstrt+1)$

werr contains the semiwidth of the uncertainty interval of the corresponding eigenvalue approximation in *w*.

spdiam

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Estimate of the spectral diameter obtained from the Gershgorin intervals

clgap1, clgapr

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Absolute gap on each end of the cluster.

Set by the calling routine to protect against shifts too close to eigenvalues outside the cluster.

pivmin

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

The minimum pivot allowed in the sturm sequence.

work

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Workspace array, dimension $(2*n)$

OUTPUT Parameters

wgap

Contains refined values of its input approximations. Very small gaps are unchanged.

sigma

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

The shift (σ) used to form $L_+ D_+ L_+^T$.

dplus

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Array, dimension (n)

The n diagonal elements of the diagonal matrix D_+ .

lplus

REAL for slarrf2

DOUBLE PRECISION for dlarrf2

Array, dimension ($n-1$)

The first $(n-1)$ elements of *lplus* contain the subdiagonal elements of the unit bidiagonal matrix L_+ .

?larrv2

Computes the eigenvectors of the tridiagonal matrix $T = L^* D^* L^T$ given L , D and the eigenvalues of $L^* D^* L^T$.

Syntax

Fortran:

```
call slarrv2( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, needil, neediu, minrgp,
rtol1, rtol2, w, werr, wgap, iblock, indexw, gers, sdiam, z, ldz, isuppz, work, iwork,
vstart, finish, maxcls, ndepth, parity, zoffset, info )
call dlarrv2( n, vl, vu, d, l, pivmin, isplit, m, dol, dou, needil, neediu, minrgp,
rtol1, rtol2, w, werr, wgap, iblock, indexw, gers, sdiam, z, ldz, isuppz, work, iwork,
vstart, finish, maxcls, ndepth, parity, zoffset, info )
```

Include Files

- C: mkl_scalapack.h

Description

?larrv2 computes the eigenvectors of the tridiagonal matrix $T = L^* D^* L^T$ given L , D and approximations to the eigenvalues of $L^* D^* L^T$. The input eigenvalues should have been computed by ?larre2a or by previous calls to ?larrv2.

The major difference between the parallel and the sequential construction of the representation tree is that in the parallel case, not all eigenvalues of a given cluster might be computed locally. Other processors might "own" and refine part of an eigenvalue cluster. This is crucial for scalability. Thus there might be communication necessary before the current level of the representation tree can be parsed.

Please note:

- The calling sequence has two additional INTEGER parameters, *dol* and *dou*, that should satisfy $m \geq dou \geq dol \geq 1$. These parameters are only relevant when both eigenvalues and eigenvectors are computed (stegr2b parameter *jobz* = 'V'). ?larrv2 only computes the eigenvectors corresponding to eigenvalues *dol* through *dou* in *w*. (That is, instead of computing the eigenvectors belonging to *w*(1) through *w*(*m*), only the eigenvectors belonging to eigenvalues *w*(*dol*) through *w*(*dou*) are computed. In this case, only the eigenvalues *dol:dou* are guaranteed to be accurately refined to all figures by Rayleigh-Quotient iteration.

- The additional arguments `vstart`, `finish`, `ndepth`, `parity`, `zoffset` are included as a thread-safe implementation equivalent to save variables. These variables store details about the local representation tree which is computed layerwise. For scalability reasons, eigenvalues belonging to the locally relevant representation tree might be computed on other processors. These need to be communicated before the inspection of the RRRs can proceed on any given layer. Note that only when the variable `finish` is true, the computation has ended. All eigenpairs between `d0l` and `d0u` have been computed. `m` is set to `d0u - d0l + 1`.
- ?larrv2 needs more workspace in `z` than the sequential SLARRV. It is used to store the conformal embedding of the local representation tree.

Input Parameters

`n`

INTEGER

The order of the matrix. $n \geq 0$.`vl, vu`

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Lower and upper bounds of the interval that contains the desired eigenvalues. $vl < vu$. Needed to compute gaps on the left or right end of the extremal eigenvalues in the desired range. `vu` is currently not used but kept as parameter in case needed.

`d`

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array, dimension (n)The n diagonal elements of the diagonal matrix `d`. On exit, `d` is overwritten.`l`

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array, dimension (n)

The $(n-1)$ subdiagonal elements of the unit bidiagonal matrix L are in elements 1 to $n-1$ of `l` (if the matrix is not split.) At the end of each block is stored the corresponding shift as given by ?larre. On exit, `l` is overwritten.

`pivmin`

DOUBLE PRECISION

The minimum pivot allowed in the sturm sequence.

`isplit`INTEGER array, dimension (n)

The splitting points, at which T breaks up into blocks. The first block consists of rows/columns 1 to `isplit(1)`, the second of rows/columns `isplit(1) + 1` through `isplit(2)`, etc.

`m`

INTEGER

The total number of input eigenvalues. $0 \leq m \leq n$.`d0l, d0u`

INTEGER

If the user wants to compute only selected eigenvectors from all the eigenvalues supplied, he can specify an index range $dol:dou$. Or else the setting $dol=1, dou=m$ should be applied. Note that dol and dou refer to the order in which the eigenvalues are stored in w . If the user wants to compute only selected eigenpairs, then the columns $dol-1$ to $dou+1$ of the eigenvector space z contain the computed eigenvectors. All other columns of z are set to zero.

If $dol > 1$, then $z(:,dol-1-zoffset)$ is used.

If $dou < m$, then $z(:,dou+1-zoffset)$ is used.

needil, neediu

INTEGER

Describe which are the left and right outermost eigenvalues that still need to be included in the computation. These indices indicate whether eigenvalues from other processors are needed to correctly compute the conformally embedded representation tree.

When $dol \leq needil \leq neediu \leq dou$, all required eigenvalues are local to the processor and no communication is required to compute its part of the representation tree.

minrgp

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

The minimum relative gap threshold to decide whether an eigenvalue or a cluster boundary is reached.

rto11, rto12

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Parameters for bisection. An interval $[left,right]$ has converged if $right-left < \max(rto11*gap, rto12*\max(|left|,|right|))$

w

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array, dimension (n)

The first m elements of w contain the approximate eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array w from ?stegr2a is expected here.) Furthermore, they are with respect to the shift of the corresponding root representation for their block.

werr

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array, dimension (n)

The first m elements contain the semiwidth of the uncertainty interval of the corresponding eigenvalue in w .

wgap

REAL for slarrv2

DOUBLE PRECISION for dlarrv2

	Array, dimension (n)
	The separation from the right neighbor eigenvalue in w .
<i>iblock</i>	INTEGER Array, dimension (n)
	The indices of the blocks (submatrices) associated with the corresponding eigenvalues in w ; $iblock(i)=1$ if eigenvalue $w(i)$ belongs to the first block from the top, $iblock(i)=2$ if $w(i)$ belongs to the second block, etc.
<i>indexw</i>	INTEGER Array, dimension (n)
	The indices of the eigenvalues within each block (submatrix); for example, $indexw(i)=10$ and $iblock(i)=2$ imply that the i -th eigenvalue $w(i)$ is the 10th eigenvalue in the second block.
<i>gers</i>	REAL for slarrv2 DOUBLE PRECISION for dlarrv2
	Array, dimension (2^*n)
	The n Gershgorin intervals (the i -th Gershgorin interval is $(gers(2*i-1), gers(2*i))$). The Gershgorin intervals should be computed from the original unshifted matrix.
	Not used but kept as parameter for possible future use.
<i>sdiam</i>	REAL for slarrv2 DOUBLE PRECISION for dlarrv2
	Array, dimension (n)
	The spectral diameters for all unreduced blocks.
<i>ldz</i>	INTEGER
	The leading dimension of the array z . $ldz \geq 1$, and if <code>stegr2b</code> parameter $jobz = 'V'$, $ldz \geq \max(1,n)$.
<i>work</i>	REAL for slarrv2 DOUBLE PRECISION for dlarrv2
	(workspace) array, dimension (12^*n)
<i>iwork</i>	(workspace) INTEGER Array, dimension (7^*n)
<i>vstart</i>	LOGICAL .TRUE. on initialization, set to .FALSE. afterwards.
<i>finish</i>	LOGICAL A flag that indicates whether all eigenpairs have been computed.
<i>maxcls</i>	INTEGER The largest cluster worked on by this processor in the representation tree.
<i>ndepth</i>	INTEGER

The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.

<i>parity</i>	INTEGER
	An internal parameter needed for the storage of the clusters on the current level of the representation tree.
<i>zoffset</i>	INTEGER

Offset for storing the eigenpairs when *z* is distributed in 1D-cyclic fashion.

OUTPUT Parameters

needil, *neediu*

w Unshifted eigenvalues for which eigenvectors have already been computed.

werr Contains refined values of its input approximations.

wgap Contains refined values of its input approximations. Very small gaps are changed.

z REAL for slarrv2

DOUBLE PRECISION for dlarrv2

Array, dimension (*ldz*, max(1,*m*))

If *info* = 0, the first *m* columns of *z* contain the orthonormal eigenvectors of the matrix *T* corresponding to the input eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w(i)*.

In the distributed version, only a subset of columns is accessed, see *dol*, *dou*, and *zoffset*.

isuppz

INTEGER array, dimension (2*max(1,*m*))

The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th eigenvector is nonzero only in elements *isuppz*(2*i-1) through *isuppz*(2*i).

vstart

.TRUE. on initialization, set to .FALSE. afterwards.

finish

A flag that indicates whether all eigenpairs have been computed.

maxcls

The largest cluster worked on by this processor in the representation tree.

ndepth

The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.

parity

An internal parameter needed for the storage of the clusters on the current level of the representation tree.

info

INTEGER

= 0: successful exit

> 0: A problem occurred in ?larrv2.

< 0: One of the called subroutines signaled an internal problem.

Needs inspection of the corresponding parameter *info* for further information.

=-1: Problem in `?larrb2` when refining a child's eigenvalues.

=-2: Problem in `?larrf2` when computing the RRR of a child. When a child is inside a tight cluster, it can be difficult to find an RRR. A partial remedy from the user's point of view is to make the parameter *minrgp* smaller and recompile. However, as the orthogonality of the computed vectors is proportional to $1/\text{minrgp}$, be aware that decreasing *minrgp* might be reduce precision.

=-3: Problem in `?larrb2` when refining a single eigenvalue after the Rayleigh correction was rejected.

= 5: The Rayleigh Quotient Iteration failed to converge to full accuracy.

?lasorte

Sorts eigenpairs by real and complex data types.

Syntax

Fortran:

```
call slasorte(s, lds, j, out, info)
call dlasorte(s, lds, j, out, info)
```

C:

```
void slasorte (float *s , MKL_INT *lds , MKL_INT *j , float *out , MKL_INT *info );
void dlasorte (double *s , MKL_INT *lds , MKL_INT *j , double *out , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `?lasorte` routine sorts eigenpairs so that real eigenpairs are together and complex eigenpairs are together. This helps to employ 2x2 shifts easily since every second subdiagonal is guaranteed to be zero. This routine does no parallel work and makes no calls.

Input Parameters

s (local) INTEGER.
 REAL for `slasorte`
 DOUBLE PRECISION for `dlasorte`
 Array, size (*lds*).
 On entry, a matrix already in Schur form.

lds (local) INTEGER.
 On entry, the leading dimension of the array *s*; unchanged on exit.

j (local) INTEGER.

On entry, the order of the matrix S ; unchanged on exit.

out

(local) INTEGER.

REAL for slasorte

DOUBLE PRECISION for dlasorte

Array, size $(2*j)$. The work buffer required by the routine.

info

(local) INTEGER.

Set, if the input matrix had an odd number of real eigenvalues and things could not be paired or if the input matrix S was not originally in Schur form. 0 indicates successful completion.

Output Parameters

s

On exit, the diagonal blocks of S have been rewritten to pair the eigenvalues. The resulting matrix is no longer similar to the input.

out

Work buffer.

?lasrt2

Sorts numbers in increasing or decreasing order.

Syntax

Fortran:

```
call slasrt2(id, n, d, key, info)
call dlasrt2(id, n, d, key, info)
```

C:

```
void slasrt2 (char *id , MKL_INT *n , float *d , MKL_INT *key , MKL_INT *info );
void dlasrt2 (char *id , MKL_INT *n , double *d , MKL_INT *key , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The ?lasrt2 routine is modified LAPACK routine [?lasrt](#), which sorts the numbers in d in increasing order (if $id = 'I'$) or in decreasing order (if $id = 'D'$). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20 . Dimension of STACK limits n to about 2^{32} .

Input Parameters

id

CHARACTER*1.

= 'I': sort d in increasing order;

= 'D': sort d in decreasing order.

n

INTEGER. The length of the array d .

d

REAL for slasrt2

DOUBLE PRECISION for dlasrt2.

Array, size (n).

On entry, the array to be sorted.

key INTEGER.

Array, size (n).

On entry, *key* contains a key to each of the entries in $d()$.

Typically, $key(i) = i$ for all i .

Output Parameters

D On exit, *d* has been sorted into increasing order
 $(d(1) \leq \dots \leq d(n))$ or into decreasing order
 $(d(1) \geq \dots \geq d(n))$, depending on *id*.

info INTEGER.
= 0: successful exit
< 0: if *info* = - i , the i -th argument had an illegal value.

key On exit, *key* is permuted in exactly the same manner as *d()* was permuted from input to output. Therefore, if $key(i) = i$ for all i upon input, then $d_{\text{out}}(i) = d_{\text{in}}(key(i))$.

?stegr2

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

Fortran:

```
call sstegr2( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, work,
lwork, iwork, liwork, dol, zoffset, info )
call dstegr2( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, isuppz, work,
lwork, iwork, liwork, dol, zoffset, info )
```

Include Files

- C: mkl_scalapack.h

Description

?stegr2 computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix T . It is invoked in the ScaLAPACK MRRR driver p?syevr and the corresponding Hermitian version either when only eigenvalues are to be computed, or when only a single processor is used (the sequential-like case).

?stegr2 has been adapted from LAPACK's ?stegr. Please note the following crucial changes.

1. The calling sequence has two additional INTEGER parameters, *dol* and *dou*, that should satisfy $m \geq dou \geq dol \geq 1$. ?steqr2 only computes the eigenpairs corresponding to eigenvalues *dol* through *dou* in *w*. (That is, instead of computing the eigenpairs belonging to *w*(1) through *w*(*m*), only the eigenvectors belonging to eigenvalues *w*(*dol*) through *w*(*dou*) are computed. In this case, only the eigenvalues *dol:dou* are guaranteed to be fully accurate.
2. *m* is not the number of eigenvalues specified by *range*, but is $m = dou - dol + 1$. This concerns the case where only eigenvalues are computed, but on more than one processor. Thus, in this case *m* refers to the number of eigenvalues computed on this processor.
3. The arrays *w* and *z* might not contain all the wanted eigenpairs locally, instead this information is distributed over other processors.

Input Parameters

<i>jobz</i>	CHARACTER*1
	= 'N': Compute eigenvalues only;
	= 'V': Compute eigenvalues and eigenvectors.
<i>range</i>	CHARACTER*1
	= 'A': all eigenvalues will be found.
	= 'V': all eigenvalues in the half-open interval (<i>vl</i> , <i>vu</i>] will be found.
	= 'I': the <i>il</i> -th through <i>iu</i> -th eigenvalues will be found.
<i>n</i>	INTEGER
	The order of the matrix. $n \geq 0$.
<i>d</i>	REAL for ssteqr2 DOUBLE PRECISION for dstegr2 Array, dimension (<i>n</i>) On entry, the <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i> . Overwritten on exit.
<i>e</i>	REAL for ssteqr2 DOUBLE PRECISION for dstegr2 Array, dimension (<i>n</i>) On entry, the (<i>n</i> -1) subdiagonal elements of the tridiagonal matrix <i>T</i> in elements 1 to <i>n</i> -1 of <i>e</i> . <i>e</i> (<i>n</i>) need not be set on input, but is used internally as workspace. Overwritten on exit.
<i>vl</i>	REAL for ssteqr2 DOUBLE PRECISION for dstegr2
<i>vu</i>	REAL for ssteqr2 DOUBLE PRECISION for dstegr2 If <i>range</i> ='V', the lower and upper bounds of the interval to be searched for eigenvalues. <i>vl</i> < <i>vu</i> . Not referenced if <i>range</i> = 'A' or 'I'.

<i>il, iu</i>	INTEGER
	If <i>range</i> ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 ≤ <i>il</i> ≤ <i>iu</i> ≤ <i>n</i> , if <i>n</i> > 0. Not referenced if <i>range</i> = 'A' or 'V'.
<i>ldz</i>	INTEGER
	The leading dimension of the array <i>z</i> . <i>ldz</i> ≥ 1, and if <i>jobz</i> = 'V', then <i>ldz</i> ≥ $\max(1, n)$.
<i>nzc</i>	INTEGER
	The number of eigenvectors to be held in the array <i>z</i> . If <i>range</i> = 'A', then <i>nzc</i> ≥ $\max(1, n)$. If <i>range</i> = 'V', then <i>nzc</i> ≥ the number of eigenvalues in (<i>vl</i> , <i>vu</i>]. If <i>range</i> = 'I', then <i>nzc</i> ≥ <i>iu</i> - <i>il</i> +1. If <i>nzc</i> = -1, then a workspace query is assumed; the routine calculates the number of columns of the array <i>z</i> that are needed to hold the eigenvectors. This value is returned as the first entry of the <i>z</i> array, and no error message related to <i>nzc</i> is issued.
<i>lwork</i>	INTEGER
	The dimension of the array <i>work</i> . <i>lwork</i> ≥ $\max(1, 18 * n)$ if <i>jobz</i> = 'V', and <i>lwork</i> ≥ $\max(1, 12 * n)$ if <i>jobz</i> = 'N'. If <i>lwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>work</i> array, returns this value as the first entry of the <i>work</i> array, and no error message related to <i>lwork</i> is issued.
<i>liwork</i>	INTEGER
	The dimension of the array <i>iwork</i> . <i>liwork</i> ≥ $\max(1, 10 * n)$ if the eigenvectors are desired, and <i>liwork</i> ≥ $\max(1, 8 * n)$ if only the eigenvalues are to be computed. If <i>liwork</i> = -1, then a workspace query is assumed; the routine only calculates the optimal size of the <i>iwork</i> array, returns this value as the first entry of the <i>iwork</i> array, and no error message related to <i>liwork</i> is issued.
<i>dol, dou</i>	INTEGER
	From the eigenvalues <i>w</i> (1: <i>m</i>), only eigenvectors <i>z</i> (:, <i>dol</i>) to <i>z</i> (:, <i>dou</i>) are computed. If <i>dol</i> > 1, then <i>z</i> (:, <i>dol</i> -1- <i>zoffset</i>) is used and overwritten. If <i>dou</i> < <i>m</i> , then <i>z</i> (:, <i>dou</i> +1- <i>zoffset</i>) is used and overwritten.
<i>zoffset</i>	INTEGER
	Offset for storing the eigenpairs when <i>z</i> is distributed in 1D-cyclic fashion

OUTPUT Parameters

m

INTEGER

Globally summed over all processors, *m* equals the total number of eigenvalues found. $0 \leq m \leq n$. If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu-il+1*. The local output equals *m* = *dou - dol + 1*.

*w*REAL array, dimension (*n*) for `ssteqr2`DOUBLE PRECISION array, dimension (*n*) for `dstegr2`Array, dimension (*n*)

The first *m* elements contain the selected eigenvalues in ascending order. Note that immediately after exiting this routine, only the eigenvalues from position *dol:dou* are reliable on this processor because the eigenvalue computation is done in parallel. Other processors will hold reliable information on other parts of the *w* array. This information is communicated in the ScalAPACK driver.

*z*REAL for `ssteqr2`DOUBLE PRECISION for `dstegr2`Array, dimension (*ldz, max(1,m)*)

If *jobz* = 'V', and if *info* = 0, then the first *m* columns of *z* contain some of the orthonormal eigenvectors of the matrix *T* corresponding to the selected eigenvalues, with the *i*-th column of *z* holding the eigenvector associated with *w(i)*.

If *jobz* = 'N', then *z* is not referenced.

Note: the user must ensure that at least $\max(1,m)$ columns are supplied in the array *z*; if *range* = 'V', the exact value of *m* is not known in advance and can be computed with a workspace query by setting *nzc* = -1, see below.

*isuppz*INTEGER array, dimension ($2*\max(1,m)$)

The support of the eigenvectors in *z*, i.e., the indices indicating the nonzero elements in *z*. The *i*-th computed eigenvector is nonzero only in elements *isuppz(2*i-1)* through *isuppz(2*i)*. This is relevant in the case when the matrix is split. *isuppz* is only set if *n>2*.

*work*On exit, if *info* = 0, *work(1)* returns the optimal (and minimal) *lwork*.*iwork*On exit, if *info* = 0, *iwork(1)* returns the optimal *liwork*.*info*

INTEGER

On exit, *info*

= 0: successful exit

other:if *info* = *-i*, the *i*-th argument had an illegal valueif *info* = 10X, internal error in `?larre2`,if *info* = 20X, internal error in `?larrv`.

Here, the digit $X = \text{ABS}(\text{iinfo}) < 10$, where iinfo is the nonzero error code returned by `?larre2` or `?larryv`, respectively.

?stegr2a

Computes selected eigenvalues and initial representations needed for eigenvector computations.

Syntax

Fortran:

```
call sstegr2a( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, work, lwork,
iwork, liwork, dol, dou, needil, neediu, inderr, nsplit, pivmin, scale, wl, wu, info )
call dstegr2a( jobz, range, n, d, e, vl, vu, il, iu, m, w, z, ldz, nzc, work, lwork,
iwork, liwork, dol, dou, needil, neediu, inderr, nsplit, pivmin, scale, wl, wu, info )
```

Include Files

- C: mkl_scalapack.h

Description

`?stegr2a` computes selected eigenvalues and initial representations needed for eigenvector computations in `?stegr2b`. It is invoked in the ScalAPACK MRRR driver `p?syevr` and the corresponding Hermitian version when both eigenvalues and eigenvectors are computed in parallel on multiple processors. For this case, `?stegr2a` implements the first part of the MRRR algorithm, parallel eigenvalue computation and finding the root RRR. At the end of `?stegr2a`, other processors might have a part of the spectrum that is needed to continue the computation locally. Once this eigenvalue information has been received by the processor, the computation can then proceed by calling the second part of the parallel MRRR algorithm, `?stegr2b`.

Please note:

- The calling sequence has two additional INTEGER parameters, (compared to LAPACK's `?stegr`), these are `dol` and `dou` and should satisfy $m \geq dou \geq dol \geq 1$. These parameters are only relevant for the case `jobz = 'V'`.

Globally invoked over all processors, `?stegr2a` computes all the eigenvalues specified by `range`.

`?stegr2a` locally only computes the eigenvalues corresponding to eigenvalues `dol` through `dou` in `w`. (That is, instead of computing the eigenvectors belonging to `w(1)` through `w(m)`, only the eigenvectors belonging to eigenvalues `w(dol)` through `w(dou)` are computed. In this case, only the eigenvalues `dol:dou` are guaranteed to be fully accurate.

- `m` is not the number of eigenvalues specified by `range`, but it is $m = dou - dol + 1$. Instead, `m` refers to the number of eigenvalues computed on this processor.
- While no eigenvectors are computed in `?stegr2a` itself (this is done later in `?stegr2b`), the interface

If `jobz = 'V'` then, depending on `range` and `dol, dou`, `?stegr2a` might need more workspace in `z` than the original `?stegr`. In particular, the arrays `w` and `z` might not contain all the wanted eigenpairs locally, instead this information is distributed over other processors.

Input Parameters

<code>jobz</code>	CHARACTER*1
	= 'N': Compute eigenvalues only;
	= 'V': Compute eigenvalues and eigenvectors.

<i>range</i>	CHARACTER*1
	= 'A': all eigenvalues will be found.
	= 'V': all eigenvalues in the half-open interval (<i>vl</i> , <i>vu</i>] will be found.
	= 'I': the <i>il</i> -th through <i>iu</i> -th eigenvalues will be found.
<i>n</i>	INTEGER
	The order of the matrix. <i>n</i> ≥ 0.
<i>d</i>	REAL for ssteigr2a DOUBLE PRECISION for dstegr2a Array, dimension (<i>n</i>)
	The <i>n</i> diagonal elements of the tridiagonal matrix <i>T</i> . Overwritten on exit.
<i>e</i>	REAL for ssteigr2a DOUBLE PRECISION for dstegr2a Array, dimension (<i>n</i>)
	On entry, the (<i>n</i> -1) subdiagonal elements of the tridiagonal matrix <i>T</i> in elements 1 to <i>n</i> -1 of <i>e</i> . <i>e</i> (<i>n</i>) need not be set on input, but is used internally as workspace. Overwritten on exit.
<i>vl</i> , <i>vu</i>	REAL for ssteigr2a DOUBLE PRECISION for dstegr2a If <i>range</i> ='V', the lower and upper bounds of the interval to be searched for eigenvalues. <i>vl</i> < <i>vu</i> . Not referenced if <i>range</i> = 'A' or 'I'.
<i>il</i> , <i>iu</i>	INTEGER If <i>range</i> ='I', the indices (in ascending order) of the smallest and largest eigenvalues to be returned. 1 ≤ <i>il</i> ≤ <i>iu</i> ≤ <i>n</i> , if <i>n</i> > 0. Not referenced if <i>range</i> = 'A' or 'V'.
<i>ldz</i>	INTEGER The leading dimension of the array <i>z</i> . <i>ldz</i> ≥ 1, and if <i>jobz</i> = 'V', then <i>ldz</i> ≥ max(1, <i>n</i>).
<i>nzc</i>	INTEGER The number of eigenvectors to be held in the array <i>z</i> . If <i>range</i> = 'A', then <i>nzc</i> ≥ max(1, <i>n</i>). If <i>range</i> = 'V', then <i>nzc</i> ≥ the number of eigenvalues in (<i>vl</i> , <i>vu</i>]. If <i>range</i> = 'I', then <i>nzc</i> ≥ <i>iu</i> - <i>il</i> +1. If <i>nzc</i> = -1, then a workspace query is assumed; the routine calculates the number of columns of the array <i>z</i> that are needed to hold the eigenvectors. This value is returned as the first entry of the <i>z</i> array, and no error message related to <i>nzc</i> is issued.

lwork

INTEGER

The dimension of the array *work*. $lwork \geq \max(1, 18*n)$ if *jobz* = 'V', and $lwork \geq \max(1, 12*n)$ if *jobz* = 'N'.

If *lwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *work* array, returns this value as the first entry of the *work* array, and no error message related to *lwork* is issued.

liwork

INTEGER

The dimension of the array *iwork*. $liwork \geq \max(1, 10*n)$ if the eigenvectors are desired, and $liwork \geq \max(1, 8*n)$ if only the eigenvalues are to be computed.

If *liwork* = -1, then a workspace query is assumed; the routine only calculates the optimal size of the *iwork* array, returns this value as the first entry of the *iwork* array, and no error message related to *liwork* is issued.

dol, dou

INTEGER

From all the eigenvalues $w(1:m)$, only eigenvalues $w(dol:dou)$ are computed.

OUTPUT Parameters

m

INTEGER

Globally summed over all processors, *m* equals the total number of eigenvalues found. $0 \leq m \leq n$.

If *range* = 'A', *m* = *n*, and if *range* = 'I', *m* = *iu-il+1*.

The local output equals *m* = *dou - dol + 1*.

w

REAL for ssteqr2a

DOUBLE PRECISION for dstegr2a

Array, dimension (*n*)

The first *m* elements contain approximations to the selected eigenvalues in ascending order. Note that immediately after exiting this routine, only the eigenvalues from position *dol:dou* are reliable on this processor because the eigenvalue computation is done in parallel. The other entries outside *dol:dou* are very crude preliminary approximations. Other processors hold reliable information on these other parts of the *w* array.

This information is communicated in the ScaLAPACK driver.

z

REAL for ssteqr2a

DOUBLE PRECISION for dstegr2a

Array, dimension (*ldz, max(1,m)*)

?steqr2a does not compute eigenvectors, this is done in ?steqr2b. The argument *z* as well as all related other arguments only appear to keep the interface consistent and to signal to the user that this subroutine is meant to be used when eigenvectors are computed.

<i>work</i>	On exit, if <i>info</i> = 0, <i>work</i> (1) returns the optimal (and minimal) <i>lwork</i> .
<i>iwork</i>	On exit, if <i>info</i> = 0, <i>iwork</i> (1) returns the optimal <i>liwork</i> .
<i>needil</i> , <i>neediu</i>	INTEGER The indices of the leftmost and rightmost eigenvalues needed to accurately compute the relevant part of the representation tree. This information can be used to find out which processors have the relevant eigenvalue information needed so that it can be communicated.
<i>inderr</i>	INTEGER <i>inderr</i> points to the place in the work space where the eigenvalue uncertainties (errors) are stored.
<i>nsplit</i>	INTEGER The number of blocks into which <i>T</i> splits. $1 \leq nsplit \leq n$.
<i>pivmin</i>	REAL for <i>ssteqr2a</i> DOUBLE PRECISION for <i>dsteqr2a</i> The minimum pivot in the sturm sequence for <i>T</i> .
<i>scale</i>	REAL for <i>ssteqr2a</i> DOUBLE PRECISION for <i>dsteqr2a</i> The scaling factor for the tridiagonal <i>T</i> .
<i>wl</i> , <i>wu</i>	REAL for <i>ssteqr2a</i> DOUBLE PRECISION for <i>dsteqr2a</i> The interval (<i>wl</i> , <i>wu</i>] contains all the wanted eigenvalues. It is either given by the user or computed in ?larre2a .
<i>info</i>	INTEGER On exit, <i>info</i> = 0: successful exit other: if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value if <i>info</i> = 10 <i>x</i> , internal error in ?larre2a , Here, the digit <i>x</i> = <i>abs</i> (<i>iinfo</i>) < 10, where <i>iinfo</i> is the nonzero error code returned by ?larre2a .

?steqr2b

From eigenvalues and initial representations computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors.

Syntax

Fortran:

```
call sstegr2b( jobz, n, d, e, m, w, z, ldz, nzc, isuppz, work, lwork, iwork, liwork,
dol, dou, needil, neediu, indwlc, pivmin, scale, wl, wu, vstart, finish, maxcls,
ndepth, parity, zoffset, info )

call dstegr2b( jobz, n, d, e, m, w, z, ldz, nzc, isuppz, work, lwork, iwork, liwork,
dol, dou, needil, neediu, indwlc, pivmin, scale, wl, wu, vstart, finish, maxcls,
ndepth, parity, zoffset, info )
```

Include Files

- C: mkl_scalapack.h

Description

?stegr2b should only be called after a call to ?stegr2a. From eigenvalues and initial representations computed by ?stegr2a, ?stegr2b computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors. It is potentially invoked multiple times on a given processor because the locally relevant representation tree might depend on spectral information that is "owned" by other processors and might need to be communicated.

Please note:

- The calling sequence has two additional integer parameters, *dol* and *dou*, that should satisfy $m \geq dol \geq dol \geq 1$. These parameters are only relevant for the case *jobz* = 'V'. ?stegr2b only computes the eigenvectors corresponding to eigenvalues *dol* through *dou* in *w*. (That is, instead of computing the eigenvectors belonging to *w*(1) through *w*(*m*), only the eigenvectors belonging to eigenvalues *w*(*dol*) through *w*(*dou*) are computed. In this case, only the eigenvalues *dol:dou* are guaranteed to be accurately refined to all figures by Rayleigh-Quotient iteration.)
- The additional arguments *vstart*, *finish*, *ndepth*, *parity*, *zoffset* are included as a thread-safe implementation equivalent to save variables. These variables store details about the local representation tree which is computed layerwise. For scalability reasons, eigenvalues belonging to the locally relevant representation tree might be computed on other processors. These need to be communicated before the inspection of the RRRs can proceed on any given layer. Note that only when the variable *finish* is true, the computation has ended. All eigenpairs between *dol* and *dou* have been computed. *m* is set to *dou* - *dol* + 1.
- ?stegr2b needs more workspace in *z* than the sequential ?stegr. It is used to store the conformal embedding of the local representation tree.

Input Parameters

<i>jobz</i>	CHARACTER*1
	= 'N': Compute eigenvalues only;
	= 'V': Compute eigenvalues and eigenvectors.
<i>n</i>	INTEGER
	The order of the matrix. <i>n</i> ≥ 0 .
<i>d</i>	REAL for sstegr2b DOUBLE PRECISION for dstegr2b Array, dimension (<i>n</i>) The <i>n</i> diagonal elements of the tridiagonal matrix T. Overwritten on exit.
<i>e</i>	REAL for sstegr2b DOUBLE PRECISION for dstegr2b

Array, dimension (n)

The ($n-1$) subdiagonal elements of the tridiagonal matrix T in elements 1 to $n-1$ of e . $e(n)$ need not be set on input, but is used internally as workspace. Overwritten on exit.

m

INTEGER

The total number of eigenvalues found in ?steqr2a. $0 \leq m \leq n$.

w

REAL for ssteqr2b

DOUBLE PRECISION for dstegr2b

Array, dimension (n)

The first m elements contain approximations to the selected eigenvalues in ascending order. Note that only the eigenvalues from the locally relevant part of the representation tree, that is all the clusters that include eigenvalues from $dol:dou$, are reliable on this processor. (It does not need to know about any others anyway.)

ldz

INTEGER

The leading dimension of the array z . $ldz \geq 1$, and if $jobz = 'V'$, then $ldz \geq \max(1,n)$.

nzc

INTEGER

The number of eigenvectors to be held in the array z .

$lwork$

INTEGER

The dimension of the array $work$. $lwork \geq \max(1,18*n)$

if $jobz = 'V'$, and $lwork \geq \max(1,12*n)$ if $jobz = 'N'$.

If $lwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued.

$liwork$

INTEGER

The dimension of the array $iwork$. $liwork \geq \max(1,10*n)$ if the eigenvectors are desired, and $liwork \geq \max(1,8*n)$ if only the eigenvalues are to be computed.

If $liwork = -1$, then a workspace query is assumed; the routine only calculates the optimal size of the $iwork$ array, returns this value as the first entry of the $iwork$ array, and no error message related to $liwork$ is issued.

dol, dou

INTEGER

From the eigenvalues $w(1:m)$, only eigenvectors $z(:,dol)$ to $z(:,dou)$ are computed.

If $dol > 1$, then $z(:,dol-1-zoffset)$ is used and overwritten.

If $dou < m$, then $z(:,dou+1-zoffset)$ is used and overwritten.

$needil, neediu$

INTEGER

Describes which are the left and right outermost eigenvalues still to be computed. Initially computed by [?larre2a](#), modified in the course of the algorithm.

pivmin

REAL for sstegr2b

DOUBLE PRECISION for dstegr2b

The minimum pivot in the sturm sequence for T .*scale*

REAL for sstegr2b

DOUBLE PRECISION for dstegr2b

The scaling factor for T . Used for unscaling the eigenvalues at the very end of the algorithm.*wl, wu*

REAL for sstegr2b

DOUBLE PRECISION for dstegr2b

The interval $(wl, wu]$ contains all the wanted eigenvalues.*vstart*

LOGICAL

.TRUE. on initialization, set to .FALSE. afterwards.

finish

LOGICAL

Indicates whether all eigenpairs have been computed.

maxcls

INTEGER

The largest cluster worked on by this processor in the representation tree.

ndepth

INTEGER

The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.

parity

INTEGER

An internal parameter needed for the storage of the clusters on the current level of the representation tree.

zoffset

INTEGER

Offset for storing the eigenpairs when z is distributed in 1D-cyclic fashion.

OUTPUT Parameters

z

REAL for sstegr2b

DOUBLE PRECISION for dstegr2b

Array, dimension ($ldz, \max(1,m)$)

If $jobz = 'V'$, and if $info = 0$, then a subset of the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i -th column of z holding the eigenvector associated with $w(i)$.

See $d01$, $d02$ for more information.

<i>isuppz</i>	INTEGER array, dimension (2*max(1, <i>m</i>)) The support of the eigenvectors in <i>z</i> , i.e., the indices indicating the nonzero elements in <i>z</i> . The <i>i</i> -th computed eigenvector is nonzero only in elements <i>isuppz</i> (2*i-1) through <i>isuppz</i> (2*i). This is relevant in the case when the matrix is split. <i>isuppz</i> is only set if <i>n</i> >2.
<i>work</i>	On exit, if <i>info</i> = 0, <i>work</i> (1) returns the optimal (and minimal) <i>lwork</i> .
<i>iwork</i>	On exit, if <i>info</i> = 0, <i>iwork</i> (1) returns the optimal <i>liwork</i> .
<i>needil</i> , <i>neediu</i>	Modified in the course of the algorithm.
<i>indwl</i> c	REAL for <i>ssteigr2b</i> DOUBLE PRECISION for <i>dsteigr2b</i> Pointer into the workspace location where the local eigenvalue representations are stored. ("Local eigenvalues" are those relative to the individual shifts of the RRRs.)
<i>vstart</i>	.TRUE. on initialization, set to .FALSE. afterwards.
<i>finish</i>	Indicates whether all eigenpairs have been computed
<i>maxcls</i>	The largest cluster worked on by this processor in the representation tree.
<i>ndepth</i>	The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.
<i>parity</i>	An internal parameter needed for the storage of the clusters on the current level of the representation tree.
<i>info</i>	INTEGER On exit, <i>info</i> = 0: successful exit other:if <i>info</i> = - <i>i</i> , the <i>i</i> -th argument had an illegal value if <i>info</i> = 20 <i>x</i> , internal error in ?larrv2 . Here, the digit <i>x</i> = abs(<i>iinfo</i>) < 10, where <i>iinfo</i> is the nonzero error code returned by ?larrv2

?stein2

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.

Syntax

Fortran:

```
call sstein2(n, d, e, m, w, iblock, isplit, orfac, z, ldz, work, iwork, ifail, info)
call dstein2(n, d, e, m, w, iblock, isplit, orfac, z, ldz, work, iwork, ifail, info)
```

C:

```
void sstein2 (MKL_INT *n , float *d , float *e , MKL_INT *m , float *w , MKL_INT
*iblock , MKL_INT *isplit , float *orfac , float *z , MKL_INT *ldz , float *work ,
MKL_INT *iwork , MKL_INT *ifail , MKL_INT *info );
void dstein2 (MKL_INT *n , double *d , double *e , MKL_INT *m , double *w , MKL_INT
*iblock , MKL_INT *isplit , double *orfac , double *z , MKL_INT *ldz , double *work ,
MKL_INT *iwork , MKL_INT *ifail , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The `?stein2` routine is a modified LAPACK routine `?stein`. It computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter *maxits* (currently set to 5).

Input Parameters

<i>n</i>	INTEGER. The order of the matrix T ($n \geq 0$).
<i>m</i>	INTEGER. The number of eigenvectors to be found ($0 \leq m \leq n$).
<i>d</i> , <i>e</i> , <i>w</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i> (*) , size (<i>n</i>). The <i>n</i> diagonal elements of the tridiagonal matrix T . <i>e</i> (*) , size (<i>n</i>). The (<i>n</i> -1) subdiagonal elements of the tridiagonal matrix T , in elements 1 to <i>n</i> -1. <i>e</i> (<i>n</i>) need not be set. <i>w</i> (*) , size (<i>n</i>). The first <i>m</i> elements of <i>w</i> contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array <i>w</i> from <code>?stebz</code> with ORDER = 'B' is expected here). The dimension of <i>w</i> must be at least <code>max(1, n)</code> .
<i>iblock</i>	INTEGER. Array, size (<i>n</i>). The submatrix indices associated with the corresponding eigenvalues in <i>w</i> ; <i>iblock</i> (<i>i</i>) = 1, if eigenvalue <i>w</i> (<i>i</i>) belongs to the first submatrix from the top, <i>iblock</i> (<i>i</i>) = 2, if eigenvalue <i>w</i> (<i>i</i>) belongs to the second submatrix, etc. (The output array <i>iblock</i> from <code>?stebz</code> is expected here).
<i>isplit</i>	INTEGER. Array, size (<i>n</i>).

The splitting points, at which T breaks up into submatrices. The first submatrix consists of rows/columns 1 to $isplit(1)$, the second submatrix consists of rows/columns $isplit(1)+1$ through $isplit(2)$, etc. (The output array $isplit$ from ?stebz is expected here).

<i>orfac</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. <i>orfac</i> specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues which are within $orfac * T $ of each other are to be orthogonalized.
<i>ldz</i>	INTEGER. The leading dimension of the output array z ; $ldz \geq \max(1, n)$.
<i>work</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Workspace array, size $(5n)$.
<i>iwork</i>	INTEGER. Workspace array, size (n) .

Output Parameters

<i>z</i>	REAL for sstein2 DOUBLE PRECISION for dstein2 Array, size ldz by m . The computed eigenvectors. The eigenvector associated with the eigenvalue $w(i)$ is stored in the i -th column of z . Any vector that fails to converge is set to its current iterate after $maxits$ iterations.
<i>ifail</i>	INTEGER. Array, size (m) . On normal exit, all elements of <i>ifail</i> are zero. If one or more eigenvectors fail to converge after $maxits$ iterations, then their indices are stored in the array <i>ifail</i> .
<i>info</i>	INTEGER. <i>info</i> = 0, the exit is successful. <i>info</i> < 0: if <i>info</i> = $-i$, the i -th had an illegal value. <i>info</i> > 0: if <i>info</i> = i , then i eigenvectors failed to converge in $maxits$ iterations. Their indices are stored in the array <i>ifail</i> .

?dbtf2

Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).

Syntax

Fortran:

```
call sdbtf2(m, n, kl, ku, ab, ldab, info)
```

```
call ddbtf2(m, n, kl, ku, ab, ldab, info)
call cdbtf2(m, n, kl, ku, ab, ldab, info)
call zdbtf2(m, n, kl, ku, ab, ldab, info)
```

C:

```
void sdbtf2 (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , float *ab , MKL_INT
*ldab , MKL_INT *info );
void ddbtf2 (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , double *ab ,
MKL_INT *ldab , MKL_INT *info );
void cdbtf2 (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , MKL_Complex8 *ab ,
MKL_INT *ldab , MKL_INT *info );
void zdbtf2 (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , MKL_Complex16 *ab ,
MKL_INT *ldab , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The ?dbtf2 routine computes an LU factorization of a general real/complex m -by- n band matrix A without using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling [BLAS Routines and Functions](#).

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in A ($n \geq 0$).
<i>kl</i>	INTEGER. The number of sub-diagonals within the band of A ($kl \geq 0$).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of A ($ku \geq 0$).
<i>ab</i>	REAL for sdbtf2 DOUBLE PRECISION for ddbtf2 COMPLEX for cdbtf2 COMPLEX*16 for zdbtf2. Array, size <i>ldab</i> by <i>n</i> . The matrix A in band storage, in rows $kl+1$ to $2kl+ku+1$; rows 1 to kl of the array need not be set. The j -th column of A is stored in the j -th column of the array <i>ab</i> as follows: $ab(kl+ku+1+i-j, j) = A(i, j)$ for $\max(1, j-ku) \leq i \leq \min(m, j+kl)$.
<i>ldab</i>	INTEGER. The leading dimension of the array <i>ab</i> . ($ldab \geq 2kl + ku + 1$)

Output Parameters

ab

On exit, details of the factorization: U is stored as an upper triangular band matrix with k_l+ku superdiagonals in rows 1 to k_l+ku+1 , and the multipliers used during the factorization are stored in rows k_l+ku+2 to $2*k_l+ku+1$. See the *Application Notes* below for further details.

info

INTEGER.

= 0: successful exit

< 0: if *info* = - *i*, the *i*-th argument had an illegal value,

> 0: if *info* = + *i*, $u(i,i)$ is 0. The factorization has been completed, but the factor U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when $m = n = 6$, $k_l = 2$, $ku = 1$:

Original Matrix	LU Factorization
.....
.....
.....
.....
.....
.....

The routine does not use array elements marked *; elements marked + need not be set on entry, but the routine requires them to store elements of U , because of fill-in resulting from the row interchanges.

?dbtrf

Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).

Syntax

Fortran:

```
call sdbtrf(m, n, kl, ku, ab, ldab, info)
call ddbtrf(m, n, kl, ku, ab, ldab, info)
call cdbtrf(m, n, kl, ku, ab, ldab, info)
call zdbtrf(m, n, kl, ku, ab, ldab, info)
```

C:

```
void sdbtrf (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , float *ab , MKL_INT *ldab , MKL_INT *info );
void ddbtrf (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , double *ab ,
MKL_INT *ldab , MKL_INT *info );
void cdbtrf (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , MKL_Complex8 *ab ,
MKL_INT *ldab , MKL_INT *info );
void zdbtrf (MKL_INT *m , MKL_INT *n , MKL_INT *kl , MKL_INT *ku , MKL_Complex16 *ab ,
MKL_INT *ldab , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

This routine computes an LU factorization of a real m -by- n band matrix A without using partial pivoting or row interchanges.

This is the blocked version of the algorithm, calling [BLAS Routines and Functions](#).

Input Parameters

<i>m</i>	INTEGER. The number of rows of the matrix A ($m \geq 0$).
<i>n</i>	INTEGER. The number of columns in A ($n \geq 0$).
<i>k1</i>	INTEGER. The number of sub-diagonals within the band of A ($k1 \geq 0$).
<i>ku</i>	INTEGER. The number of super-diagonals within the band of A ($ku \geq 0$).
<i>ab</i>	REAL for sdbtrf DOUBLE PRECISION for ddbtrf COMPLEX for cdbtrf COMPLEX*16 for zdbtrf. Array, size l_{dab} by n . The matrix A in band storage, in rows $k1+1$ to $2k1+ku+1$; rows 1 to $k1$ of the array need not be set. The j -th column of A is stored in the j -th column of the array <i>ab</i> as follows: $ab(k1+ku+1+i-j, j) = A(i, j)$ for $\max(1, j-ku) \leq i \leq \min(m, j+k1)$.
<i>l_{dab}</i>	INTEGER. The leading dimension of the array <i>ab</i> . $(l_{dab} \geq 2k1 + ku + 1)$

Output Parameters

<i>ab</i>	On exit, details of the factorization: U is stored as an upper triangular band matrix with $k1+ku$ superdiagonals in rows 1 to $k1+ku+1$, and the multipliers used during the factorization are stored in rows $k1+ku+2$ to $2*k1+ku+1$. See the <i>Application Notes</i> below for further details.
<i>info</i>	INTEGER. = 0: successful exit < 0: if <i>info</i> = $-i$, the i -th argument had an illegal value, > 0: if <i>info</i> = $+i$, $u(i, i)$ is 0. The factorization has been completed, but the factor U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when $m = n = 6$, $k1 = 2$, $ku = 1$:



The routine does not use array elements marked *.

?dttrf

Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).

Syntax

Fortran:

```
call sdttrf(n, dl, d, du, info)
call ddttrf(n, dl, d, du, info)
call cdttrf(n, dl, d, du, info)
call zdttrf(n, dl, d, du, info)
```

C:

```
void sdttrf (MKL_INT *n , float *dl , float *d , float *du , MKL_INT *info );
void ddttrf (MKL_INT *n , double *dl , double *d , double *du , MKL_INT *info );
void cdttrf (MKL_INT *n , MKL_Complex8 *dl , MKL_Complex8 *d , MKL_Complex8 *du ,
MKL_INT *info );
void zdttrf (MKL_INT *n , MKL_Complex16 *dl , MKL_Complex16 *d , MKL_Complex16 *du ,
MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The ?dttrf routine computes an LU factorization of a real or complex tridiagonal matrix A using elimination without partial pivoting.

The factorization has the form $A = L \cdot U$, where L is a product of unit lower bidiagonal matrices and U is upper triangular with nonzeros only in the main diagonal and first superdiagonal.

Input Parameters

<i>n</i>	INTEGER. The order of the matrix A ($n \geq 0$).
<i>dl, d, du</i>	REAL for sdttrf DOUBLE PRECISION for ddttrf COMPLEX for cdttrf COMPLEX*16 for zdttrf. Arrays containing elements of A .

The array dl of size $(n - 1)$ contains the sub-diagonal elements of A .

The array d of size n contains the diagonal elements of A .

The array du of size $(n - 1)$ contains the super-diagonal elements of A .

Output Parameters

dl	Overwritten by the $(n-1)$ multipliers that define the matrix L from the LU factorization of A .
d	Overwritten by the n diagonal elements of the upper triangular matrix U from the LU factorization of A .
du	Overwritten by the $(n-1)$ elements of the first super-diagonal of U .
$info$	<p>INTEGER.</p> <p>= 0: successful exit</p> <p>< 0: if $info = - i$, the i-th argument had an illegal value,</p> <p>> 0: if $info = i$, $u(i,i)$ is exactly 0. The factorization has been completed, but the factor U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.</p>

?dttrs

Solves a general tridiagonal system of linear equations using the LU factorization computed by ?dttrf.

Syntax

Fortran:

```
call sdtrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call ddtrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call cdtrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
call zdtrsv(uplo, trans, n, nrhs, dl, d, du, b, ldb, info)
```

C:

```
void sdtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , float *dl ,
float *d , float *du , float *b , MKL_INT *ldb , MKL_INT *info );
void ddtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , double *dl ,
double *d , double *du , double *b , MKL_INT *ldb , MKL_INT *info );
void cdtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8
*dl , MKL_Complex8 *d , MKL_Complex8 *du , MKL_Complex8 *b , MKL_INT *ldb , MKL_INT
*info );
void zdtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16
*dl , MKL_Complex16 *d , MKL_Complex16 *du , MKL_Complex16 *b , MKL_INT *ldb , MKL_INT
*info );
```

Include Files

- C: mkl_scalapack.h

Description

The ?dttrs_V routine solves one of the following systems of linear equations:

$$L^*X = B, L^T*X = B, \text{ or } L^H*X = B,$$

$$U^*X = B, U^T*X = B, \text{ or } U^H*X = B$$

with factors of the tridiagonal matrix A from the LU factorization computed by [?dttrf](#).

Input Parameters

uplo

CHARACTER*1.

Specifies whether to solve with L or U .

trans

CHARACTER. Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If *trans* = 'N', then $A^*X=B$ is solved for X (no transpose).

If *trans* = 'T', then $A^T*X = B$ is solved for X (transpose).

If *trans* = 'C', then $A^H*X = B$ is solved for X (conjugate transpose).

n

INTEGER. The order of the matrix A ($n \geq 0$).

nrhs

INTEGER. The number of right-hand sides, that is, the number of columns in the matrix B ($nrhs \geq 0$).

dl,d,du,b

REAL for sdttrs_V

DOUBLE PRECISION for ddttrs_V

COMPLEX for cdtrs_V

COMPLEX*16 for zdtrs_V.

Arrays of sizes: $dl(n - 1)$, $d(n)$, $du(n - 1)$, $b(ldb, nrhs)$.

The array *dl* contains the $(n - 1)$ multipliers that define the matrix L from the LU factorization of A .

The array *d* contains n diagonal elements of the upper triangular matrix U from the LU factorization of A .

The array *du* contains the $(n - 1)$ elements of the first super-diagonal of U .

On entry, the array *b* contains the right-hand side matrix B .

ldb

INTEGER. The leading dimension of the array *b*; $ldb \geq \max(1, n)$.

Output Parameters

b

Overwritten by the solution matrix X .

info

INTEGER. If *info*=0, the execution is successful.

If *info* = $-i$, the *i*-th parameter had an illegal value.

?pttrsV

*Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the $L^*D^*L^H$ factorization computed by ?pttrf.*

Syntax

Fortran:

```
call spttrsv(trans, n, nrhs, d, e, b, ldb, info)
call dpttrsv(trans, n, nrhs, d, e, b, ldb, info)
call cpttrsv(uplo, trans, n, nrhs, d, e, b, ldb, info)
call zpttrsv(uplo, trans, n, nrhs, d, e, b, ldb, info)
```

C:

```
void spttrsv (char *trans , MKL_INT *n , MKL_INT *nrhs , float *d , float *e , float *b , MKL_INT *ldb , MKL_INT *info );
void dpttrsv (char *trans , MKL_INT *n , MKL_INT *nrhs , double *d , double *e , double *b , MKL_INT *ldb , MKL_INT *info );
void cpttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , float *d , MKL_Complex8 *e , MKL_Complex8 *b , MKL_INT *ldb , MKL_INT *info );
void zpttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , double *d , MKL_Complex16 *e , MKL_Complex16 *b , MKL_INT *ldb , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The ?pttrsV routine solves one of the triangular systems:

$L^T X = B$, or $L^H X = B$ for real flavors,

or

$L^H X = B$, or $L^H X = B$,

$U^H X = B$, or $U^H X = B$ for complex flavors,

where L (or U for complex flavors) is the Cholesky factor of a Hermitian positive-definite tridiagonal matrix A such that

$A = L^*D^*L^H$ (computed by [spttrf/dpttrf](#))

or

$A = U^H D^* U$ or $A = L^* D^* L^H$ (computed by [cpttrf/zpttrf](#)).

Input Parameters

uplo

CHARACTER*1. Must be 'U' or 'L'.

Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix A is stored and the form of the factorization:

If $uplo = 'U'$, e is the superdiagonal of U , and $A = U^H D^* U$ or $A = L^* D^* L^H$;

if $uplo = 'L'$, e is the subdiagonal of L , and $A = L^*D*L^H$.

The two forms are equivalent, if A is real.

trans

CHARACTER.

Specifies the form of the system of equations:

for real flavors:

if $trans = 'N'$: $L*X = B$ (no transpose)

if $trans = 'T'$: $L^T*X = B$ (transpose)

for complex flavors:

if $trans = 'N'$: $U*X = B$ or $L^H*X = B$ (no transpose)

if $trans = 'C'$: $U^H*X = B$ or $L^H*X = B$ (conjugate transpose).

n

INTEGER. The order of the tridiagonal matrix A . $n \geq 0$.

nrhs

INTEGER. The number of right hand sides, that is, the number of columns of the matrix B . $nrhs \geq 0$.

d

REAL array, size (n). The n diagonal elements of the diagonal matrix D from the factorization computed by [?pttrf](#).

e

COMPLEX array, size ($n-1$). The ($n-1$) off-diagonal elements of the unit bidiagonal factor U or L from the factorization computed by [?pttrf](#). See *uplo*.

b

COMPLEX array, size ldb by $nrhs$.

On entry, the right hand side matrix B .

ldb

INTEGER.

The leading dimension of the array b .

$ldb \geq \max(1, n)$.

Output Parameters

b

On exit, the solution matrix X .

info

INTEGER.

= 0: successful exit

< 0: if $info = -i$, the i -th argument had an illegal value.

?steqr2

Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

Syntax

Fortran:

```
call ssteqr2(compz, n, d, e, z, ldz, nr, work, info)
```

```
call dsteqr2(compz, n, d, e, z, ldz, nr, work, info)
```

C:

```
void ssteqr2 (char *compz , MKL_INT *n , float *d , float *e , float *z , MKL_INT
*ldz , MKL_INT *nr , float *work , MKL_INT *info );
void dsteqr2 (char *compz , MKL_INT *n , double *d , double *e , double *z , MKL_INT
*ldz , MKL_INT *nr , double *work , MKL_INT *info );
```

Include Files

- C: mkl_scalapack.h

Description

The ?steqr2 routine is a modified version of LAPACK routine ?steqr. The ?steqr2 routine computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. ?steqr2 is modified from ?steqr to allow each ScalAPACK process running ?steqr2 to perform updates on a distributed matrix Q. Proper usage of ?steqr2 can be gleaned from examination of ScalAPACK routine p?syev.

Input Parameters

<i>compz</i>	CHARACTER*1. Must be 'N' or 'I'. If <i>compz</i> = 'N', the routine computes eigenvalues only. If <i>compz</i> = 'I', the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix <i>T</i> .
<i>n</i>	INTEGER. The order of the matrix <i>T</i> ($n \geq 0$).
<i>d</i> , <i>e</i> , <i>work</i>	REAL for single-precision flavors DOUBLE PRECISION for double-precision flavors. Arrays: <i>d</i> contains the diagonal elements of <i>T</i> . The dimension of <i>d</i> must be at least $\max(1, n)$. <i>e</i> contains the $(n-1)$ subdiagonal elements of <i>T</i> . The dimension of <i>e</i> must be at least $\max(1, n-1)$. <i>work</i> is a workspace array. The dimension of <i>work</i> is $\max(1, 2*n-2)$. If <i>compz</i> = 'N', then <i>work</i> is not referenced.
<i>z</i>	(local) REAL for ssteqr2 DOUBLE PRECISION for dsteqr2 Array, global size <i>n</i> by <i>n</i> , local size <i>ldz</i> by <i>nr</i> . If <i>compz</i> = 'V', then <i>z</i> contains the orthogonal matrix used in the reduction to tridiagonal form.
<i>ldz</i>	INTEGER. The leading dimension of the array <i>z</i> . Constraints: $ldz \geq 1$, $ldz \geq \max(1, n)$, if eigenvectors are desired.

nr INTEGER. $nr = \max(1, numroc(n, nb, myprow, 0, nprocs))$.
If $compz = 'N'$, then nr is not referenced.

Output Parameters

d REAL array, size (n), for `ssteqr2`.
DOUBLE PRECISION array, size (n), for `dsteqr2`.
On exit, the eigenvalues in ascending order, if $info = 0$.
See also *info*.

e REAL array, size ($n-1$), for `ssteqr2`.
DOUBLE PRECISION array, size ($n-1$), for `dsteqr2`.
On exit, *e* has been destroyed.

z (local)
REAL for `ssteqr2`
DOUBLE PRECISION for `dsteqr2`
Array, global size n by n , local size ldz by nr .
On exit, if $info = 0$, then,
if $compz = 'V'$, *z* contains the orthonormal eigenvectors of the original symmetric matrix, and if $compz = 'I'$, *z* contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If $compz = 'N'$, then *z* is not referenced.

info INTEGER.
 $info = 0$, the exit is successful.
 $info < 0$: if $info = -i$, the i -th had an illegal value.
 $info > 0$: the algorithm has failed to find all the eigenvalues in a total of $30n$ iterations;
if $info = i$, then i elements of *e* have not converged to zero; on exit, *d* and *e* contain the elements of a symmetric tridiagonal matrix, which is orthogonally similar to the original matrix.

Utility Functions and Routines

This section describes ScaLAPACK utility functions and routines. Summary information about these routines is given in the following table:

ScaLAPACK Utility Functions and Routines

Routine Name	Data Types	Description
<code>p?labad</code>	<code>s, d</code>	Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.
<code>p?lachkieee</code>	<code>s, d</code>	Performs a simple check for the features of the IEEE standard. (C interface function).

Routine Name	Data Types	Description
p?lamch	s, d	Determines machine parameters for floating-point arithmetic.
p?lasnbt	s, d	Computes the position of the sign bit of a floating-point number. (C interface function).

See Also

[pxerbla](#) Error handling routine called by ScalAPACK routines.

p?labad

Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax

Fortran:

```
call pslabad(ictxt, small, large)
call pdlabad(ictxt, small, large)
```

C:

```
void pslabad (MKL_INT *ictxt , float *small , float *large );
void pdlabad (MKL_INT *ictxt , double *small , double *large );
```

Include Files

- C: mkl_scalapack.h

Description

The p?labad routine takes as input the values computed by p?lamch for underflow and overflow, and returns the square root of each of these values if the log of *large* is sufficiently large. This subroutine is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by p?lamch. This subroutine is needed because p?lamch does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

In addition, this routine performs a global minimization and maximization on these values, to support heterogeneous computing networks.

Input Parameters

<i>ictxt</i>	(global) INTEGER. The BLACS context handle in which the computation takes place.
<i>small</i>	(local). REAL PRECISION for pslabad. DOUBLE PRECISION for pdlabad. On entry, the underflow threshold as computed by p?lamch.
<i>large</i>	(local). REAL PRECISION for pslabad. DOUBLE PRECISION for pdlabad.

On entry, the overflow threshold as computed by `p?lamch`.

Output Parameters

<i>small</i>	(local).
	On exit, if $\log_{10}(large)$ is sufficiently large, the square root of <i>small</i> , otherwise unchanged.
<i>large</i>	(local).
	On exit, if $\log_{10}(large)$ is sufficiently large, the square root of <i>large</i> , otherwise unchanged.

`p?lachkieee`

Performs a simple check for the features of the IEEE standard. (C interface function).

Syntax

Fortran:

```
void pslachkieee(int *isieee, float *rmax, float *rmin);
void pdlachkieee(int *isieee, float *rmax, float *rmin);
```

C:

```
void pslachkieee (MKL_INT *isieee , float *rmax , float *rmin );
void pdlachkieee (MKL_INT *isieee , float *rmax , float *rmin );
```

Include Files

- C: `mkl_scalapack.h`

Description

The `p?lachkieee` routine performs a simple check to make sure that the features of the IEEE standard are implemented. In some implementations, `p?lachkieee` may not return.

Note that all arguments are call-by-reference so that this routine can be directly called from Fortran code. This is a ScaLAPACK internal subroutine and arguments are not checked for unreasonable values.

Input Parameters

<i>rmax</i>	(local).
	REAL for <code>pslachkieee</code>
	DOUBLE PRECISION for <code>pdlachkieee</code>
	The overflow threshold (= <code>?lamch ('O')</code>).
<i>rmin</i>	(local).
	REAL for <code>pslachkieee</code>
	DOUBLE PRECISION for <code>pdlachkieee</code>
	The underflow threshold (= <code>?lamch ('U')</code>).

Output Parameters

isieee (local). INTEGER.

On exit, *isieee* = 1 implies that all the features of the IEEE standard that we rely on are implemented. On exit, *isieee* = 0 implies that some the features of the IEEE standard that we rely on are missing.

p?lamch

Determines machine parameters for floating-point arithmetic.

Syntax

Fortran:

```
val = pslamch(ictxt, cmach)
```

```
val = pdlamch(ictxt, cmach)
```

C:

```
float pslamch (MKL_INT *ictxt , char *cmach );
```

```
double pdlamch (MKL_INT *ictxt , char *cmach );
```

Include Files

- C: mkl_scalapack.h

Description

The p?lamch routine determines single precision machine parameters.

Input Parameters

ictxt (global). INTEGER. The BLACS context handle in which the computation takes place.

cmach (global) CHARACTER*1.

Specifies the value to be returned by p?lamch:

```
= 'E' or 'e', p?lamch := eps
= 'S' or 's' , p?lamch := sfmin
= 'B' or 'b', p?lamch := base
= 'P' or 'p', p?lamch := eps*base
= 'N' or 'n', p?lamch := t
= 'R' or 'r', p?lamch := rnd
= 'M' or 'm', p?lamch := emin
= 'U' or 'u', p?lamch := rmin
= 'L' or 'l', p?lamch := emax
= 'O' or 'o', p?lamch := rmax,
```

where

```

eps = relative machine precision
sfmin = safe minimum, such that 1/sfmin does not overflow
base = base of the machine
prec = eps*base
t = number of (base) digits in the mantissa
rnd = 1.0 when rounding occurs in addition, 0.0 otherwise
emin = minimum exponent before (gradual) underflow
rmin = underflow threshold - base^(emin-1)
emax = largest exponent before overflow
rmax = overflow threshold - (base^emax) * (1-eps)

```

Output Parameters

<i>val</i>	Value returned by the routine.
------------	--------------------------------

p?lasnbt

Computes the position of the sign bit of a floating-point number. (C interface function).

Syntax

Fortran:

```
void pslasnbt(int *ieflag);
void pdlASNBT(int *ieflag);
```

C:

```
void pslasnbt (MKL_INT *ieflag );
void pdlASNBT (MKL_INT *ieflag );
```

Include Files

- C: mkl_scalapack.h

Description

The p?lasnbt routine finds the position of the signbit of a single/double precision floating point number. This routine assumes IEEE arithmetic, and hence, tests only the 32-nd bit (for single precision) or 32-nd and 64-th bits (for double precision) as a possibility for the signbit. `sizeof(int)` is assumed equal to 4 bytes.

If a compile time flag (`NO_IEEE`) indicates that the machine does not have IEEE arithmetic, `ieflag = 0` is returned.

Output Parameters

<i>ieflag</i>	INTEGER.
---------------	----------

This flag indicates the position of the signbit of any single/double precision floating point number.

ieflag = 0, if the compile time flag `NO_IEEE` indicates that the machine does not have IEEE arithmetic, or if `sizeof(int)` is different from 4 bytes.

ieflag = 1 indicates that the signbit is the 32-nd bit for a single precision routine.

In the case of a double precision routine:

ieflag = 1 indicates that the signbit is the 32-nd bit (Big Endian).

ieflag = 2 indicates that the signbit is the 64-th bit (Little Endian).

Matrix Redistribution/Copy Routines

This section describes ScaLAPACK redistribution/copy routines. Summary information about these routines is given in the following table:

ScaLAPACK Redistribution/Copy Routines

Routine Name	Data Types	Description
p?gemr2d	s, d, c, z, i	Copies a submatrix from one general rectangular matrix to another.
p?trmr2d	s, d, c, z, i	Copies a submatrix from one trapezoidal matrix to another.

See Also

[pxerbla](#) Error handling routine called by ScaLAPACK routines.

p?gemr2d

Copies a submatrix from one general rectangular matrix to another.

Syntax

Fortran:

```
call psgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pdgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pcgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pzgemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pigemr2d(m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
```

C:

```
void psgemr2d (MKL_INT *m, MKL_INT *n, float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *ictxt );
void pdgemr2d (MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT
*ictxt );
void pcgemr2d (MKL_INT *m , MKL_INT *n MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
MKL_INT *ictxt );
void pzgemr2d (MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb ,
MKL_INT *ictxt );
void pigemr2d (MKL_INT *m , MKL_INT *n , int *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca , int *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *ictxt );
```

Include Files

- C: mkl_scalapack.h

Description

The `p?gemr2d` routine copies the indicated matrix or submatrix of A to the indicated matrix or submatrix of B . It provides a truly general copy from any block cyclicly-distributed matrix or submatrix to any other block cyclicly-distributed matrix or submatrix. With `p?trmr2d`, these routines are the only ones in the ScaLAPACK library which provide inter-context operations: they can take a matrix or submatrix A in context A (distributed over process grid A) and copy it to a matrix or submatrix B in context B (distributed over process grid B).

There does not need to be a relationship between the two operand matrices or submatrices other than their global size and the fact that they are both legal block cyclicly-distributed matrices or submatrices. This means that they can, for example, be distributed across different process grids, have varying block sizes and differing matrix starting points, or be contained in different sized distributed matrices.

Take care when context A is disjoint from context B . The general rules for which parameters need to be set are:

- All calling processes must have the correct m and n .
- Processes in context A must correctly define all parameters describing A .
- Processes in context B must correctly define all parameters describing B .
- Processes which are not members of context A must pass `descA(ctxt_) = -1` and need not set other parameters describing A .
- Processes which are not members of context B must pass `descB(ctxt_) = -1` and need not set other parameters describing B .

Because of its generality, `p?gemr2d` can be used for many operations not usually associated with copy routines. For instance, it can be used to take a matrix on one process and distribute it across a process grid, or the reverse. If a supercomputer is grouped into a virtual parallel machine with a workstation, for instance, this routine can be used to move the matrix from the workstation to the supercomputer and back. In ScaLAPACK, it is called to copy matrices from a two-dimensional process grid to a one-dimensional process grid. It can be used to redistribute matrices so that distributions providing maximal performance can be used by various component libraries, as well.

Note that this routine requires an array descriptor with `dtype_ = 1`.

Input Parameters

m	(global) INTEGER. The number of rows of matrix A to be copied ($m \geq 0$).
n	(global) INTEGER. The number of columns of matrix A to be copied ($n \geq 0$).
$nrhs$	(global) INTEGER. The number of right hand sides; the number of columns of the distributed submatrix $\text{sub}(B)$ ($nrhs \geq 0$).
a	(local) REAL for <code>psgemr2d</code> DOUBLE for <code>pdgemr2d</code> COMPLEX for <code>pcgemr2d</code> DOUBLE COMPLEX for <code>pzgemr2d</code> INTEGER for <code>pigemr2d</code> .

Pointer into the local memory to array of size lld_a by $LOCc(ja+n-1)$ containing the source matrix A .

ia, ja (global) INTEGER. The row and column indices in the array A indicating the first row and the first column, respectively, of the submatrix of A) to copy. $1 \leq ia \leq total_rows_in_a - m + 1$, $1 \leq ja \leq total_columns_in_a - n + 1$.

desca (global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix A .

Only $desca(dtype__) = 1$ is supported, so $dlen_ = 9$.

If the calling process is not part of the context of A , $desca(ctxt_)$ must be equal to -1.

ib, jb (global) INTEGER. The row and column indices in the array B indicating the first row and the first column, respectively, of the submatrix B to which to copy the matrix. $1 \leq ib \leq total_rows_in_b - m + 1$, $1 \leq jb \leq total_columns_in_b - n + 1$.

descb (global and local) INTEGER array of size $dlen_$. The array descriptor for the distributed matrix B .

Only $descb(dtype__) = 1$ is supported, so $dlen_ = 9$.

If the calling process is not part of the context of B , $descb(ctxt_)$ must be equal to -1.

ictxt (global) INTEGER.

The context encompassing at least the union of all processes in context A and context B . All processes in the context $ictxt$ must call this routine, even if they do not own a piece of either matrix.

Output Parameters

b REAL for psgemr2d
 DOUBLE for pdgemr2d
 COMPLEX for pcgemr2d
 DOUBLE COMPLEX for pzgemr2d
 INTEGER for pigemr2d.
 Pointer into the local memory to array of size lld_b by $LOCc(jb+nrhs-1)$.
 Overwritten by the submatrix from A .

p?trmr2d

Copies a submatrix from one trapezoidal matrix to another.

Syntax

Fortran:

```
call pstrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pdtrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
```

```
call pctrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pztrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
call pitrmr2d(uplo, diag, m, n, a, ia, ja, desca, b, ib, jb, descb, ictxt)
```

C:

```
void pstrmr2d (char *uplo , char *diag , MKL_INT *m , MKL_INT *n , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *ictxt );

void pdtrmr2d (char *uplo , char *diag , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , MKL_INT *ictxt );

void pctrmr2d (char *uplo , char *diag , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *ictxt );

void pztrmr2d (char *uplo , char *diag , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs ,
MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b ,
MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *ictxt );

void pitrmr2d (char *uplo , char *diag , MKL_INT *m , MKL_INT *n , int *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , int *b , MKL_INT *ib , MKL_INT *jb , MKL_INT
*descb , MKL_INT *ictxt );
```

Include Files

- C: `mkl_scalapack.h`

Description

The `p?trmr2d` routine copies the indicated matrix or submatrix of A to the indicated matrix or submatrix of B . It provides a truly general copy from any block cyclicly-distributed matrix or submatrix to any other block cyclicly-distributed matrix or submatrix. With `p?gemr2d`, these routines are the only ones in the ScaLAPACK library which provide inter-context operations: they can take a matrix or submatrix A in context A (distributed over process grid A) and copy it to a matrix or submatrix B in context B (distributed over process grid B).

The `p?trmr2d` routine assumes the matrix or submatrix to be trapezoidal. Only the upper or lower part is copied, and the other part is unchanged.

There does not need to be a relationship between the two operand matrices or submatrices other than their global size and the fact that they are both legal block cyclicly-distributed matrices or submatrices. This means that they can, for example, be distributed across different process grids, have varying block sizes and differing matrix starting points, or be contained in different sized distributed matrices.

Take care when context A is disjoint from context B . The general rules for which parameters need to be set are:

- All calling processes must have the correct m and n .
- Processes in context A must correctly define all parameters describing A .
- Processes in context B must correctly define all parameters describing B .
- Processes which are not members of context A must pass `desca(ctxt_) = -1` and need not set other parameters describing A .
- Processes which are not members of context B must pass `descb(ctxt_) = -1` and need not set other parameters describing B .

Because of its generality, `p?trmr2d` can be used for many operations not usually associated with copy routines. For instance, it can be used to take a matrix on one process and distribute it across a process grid, or the reverse. If a supercomputer is grouped into a virtual parallel machine with a workstation, for

instance, this routine can be used to move the matrix from the workstation to the supercomputer and back. In ScaLAPACK, it is called to copy matrices from a two-dimensional process grid to a one-dimensional process grid. It can be used to redistribute matrices so that distributions providing maximal performance can be used by various component libraries, as well.

Note that this routine requires an array descriptor with $dtype_ = 1$.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether to copy the upper or lower part of the matrix or submatrix.
	<i>uplo</i> = 'U' Copy the upper triangular part.
	<i>uplo</i> = 'L' Copy the lower triangular part.
<i>diag</i>	(global) CHARACTER*1. Specifies whether to copy the diagonal of the matrix or submatrix.
	<i>diag</i> = 'U' Do not copy the diagonal.
	<i>diag</i> = 'N' Copy the diagonal.
<i>m</i>	(global) INTEGER. The number of rows of matrix <i>A</i> to be copied ($m \geq 0$).
<i>n</i>	(global) INTEGER. The number of columns of matrix <i>A</i> to be copied ($n \geq 0$).
<i>a</i>	(local) REAL for pstrmr2d DOUBLE for pdtrmr2d COMPLEX for pcstrmr2d DOUBLE COMPLEX for pzstrmr2d INTEGER for pitrnr2d. Pointer into the local memory to array of size <i>lld_a</i> by <i>LOCc(ja+n-1)</i> containing the source matrix <i>A</i> .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the array <i>A</i> indicating the first row and the first column, respectively, of the submatrix of <i>A</i> to copy. $1 \leq ia \leq total_rows_in_a - m + 1$, $1 \leq ja \leq total_columns_in_a - n + 1$.
<i>desca</i>	(global and local) INTEGER array of size <i>dlen</i> . The array descriptor for the distributed matrix <i>A</i> . Only <i>desca(dtype_)</i> = 1 is supported, so <i>dlen_</i> = 9. If the calling process is not part of the context of <i>A</i> , <i>desca(ctxt_)</i> must be equal to -1.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the array <i>B</i> indicating the first row and the first column, respectively, of the submatrix <i>B</i> to which to copy the matrix. $1 \leq ib \leq total_rows_in_b - m + 1$, $1 \leq jb \leq total_columns_in_b - n + 1$.

<i>descb</i>	(global and local) INTEGER array of size <i>dlen_</i> . The array descriptor for the distributed matrix <i>B</i> . Only <i>descb(dtype_)</i> = 1 is supported, so <i>dlen_</i> = 9. If the calling process is not part of the context of <i>B</i> , <i>descb(ctxt_)</i> must be equal to -1.
<i>ictxt</i>	(global) INTEGER. The context encompassing at least the union of all processes in context <i>A</i> and context <i>B</i> . All processes in the context <i>ictxt</i> must call this routine, even if they do not own a piece of either matrix.

Output Parameters

<i>b</i>	REAL for pstrmr2d DOUBLE for pdtrmr2d COMPLEX for pcstrmr2d DOUBLE COMPLEX for pzstrmr2d INTEGER for pitrnr2d. Pointer into the local memory to array of size <i>lld_b</i> by <i>LOCc(jb+nrhs-1)</i> . Overwritten by the submatrix from <i>A</i> .
----------	---

Sparse Solver Routines

Intel® Math Kernel Library (Intel® MKL) provides user-callable sparse solver software to solve real or complex, symmetric, structurally symmetric or nonsymmetric, positive definite, indefinite or Hermitian square sparse linear system of algebraic equations.

The terms and concepts required to understand the use of the Intel MKL sparse solver routines are discussed in the [Appendix A "Linear Solvers Basics"](#). If you are familiar with linear sparse solvers and sparse matrix storage schemes, you can skip these sections and go directly to the interface descriptions.

This chapter describes

- the direct sparse solver based on PARDISO* which is referred to here as [Intel MKL PARDISO](#);
- the alternative interface for the direct sparse solver which is referred to here as the [DSS interface](#);
- [iterative sparse solvers \(ISS\)](#) based on the reverse communication interface (RCI);
- and [two preconditioners](#) based on the incomplete LU factorization technique.

Intel MKL PARDISO – Parallel Direct Sparse Solver Interface

This section describes the interface to the shared-memory multiprocessing parallel direct sparse solver known as the Intel MKL PARDISO solver. The interface is Fortran, but it can be called from C or C++ programs by observing Fortran parameter passing and naming conventions used by the supported compilers and operating systems.

The Intel MKL PARDISO package is a high-performance, robust, memory efficient, and easy to use software package for solving large sparse linear systems of equations on shared memory multiprocessors. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques [[Schenk00-2](#)]. To improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update and pipelining parallelism is used with a combination of left- and right-looking supernode techniques [[Schenk00](#), [Schenk01](#), [Schenk02](#), [Schenk03](#)]. The parallel pivoting methods allow complete supernode pivoting to compromise numerical stability and scalability during the factorization process. For sufficiently large problem sizes, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory multiprocessing architecture.

The following table lists the names of the Intel MKL PARDISO routines and describes their general use.

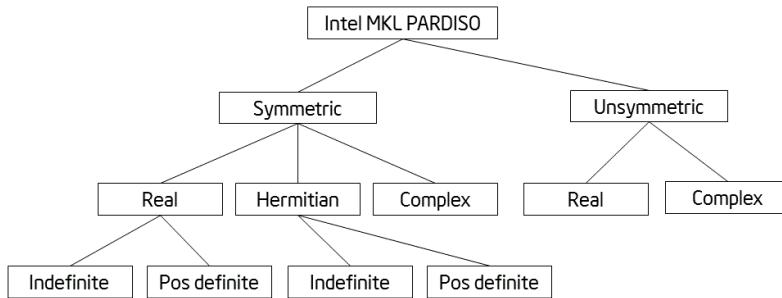
Intel MKL PARDISO Routines

Routine	Description
pardisoinit	Initializes Intel MKL PARDISO with default parameters depending on the matrix type.
pardiso	Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.
pardiso_64	Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides, 64-bit integer version.
pardiso_getenv	Retrieves additional values from the Intel MKL PARDISO handle.
pardiso_setenv	Sets additional values in the Intel MKL PARDISO handle.
mkl_pardiso_pivot	Replaces routine which handles Intel MKL PARDISO pivots with user-defined routine.

Routine	Description
<code>pardiso_getdiag</code>	Returns diagonal elements of initial and factorized matrix.
<code>pardiso_handle_store</code>	Store internal structures from <code>pardiso</code> to a file.
<code>pardiso_handle_restore</code>	Restore <code>pardiso</code> internal structures from a file.
<code>pardiso_handle_delete</code>	Delete files with <code>pardiso</code> internal structure data.
<code>pardiso_handle_store_64</code>	Store internal structures from <code>pardiso_64</code> to a file.
<code>pardiso_handle_restore_64</code>	Restore <code>pardiso_64</code> internal structures from a file.
<code>pardiso_handle_delete_64</code>	Delete files with <code>pardiso_64</code> internal structure data.

The Intel MKL PARDISO solver supports a wide range of real and complex sparse matrix types (see the figure below).

Sparse Matrices That Can Be Solved with the Intel MKL PARDISO Solver



The Intel MKL PARDISO solver performs four tasks:

- analysis and symbolic factorization
- numerical factorization
- forward and backward substitution including iterative refinement
- termination to release all internal solver memory.

You can find code examples that use Intel MKL PARDISO routines to solve systems of linear equations in the `examples` folder of the Intel MKL installation directory:

- Fortran examples: `examples/solverf/source`
- C examples: `examples/solverc/source`

Supported Matrix Types

The analysis steps performed by Intel MKL PARDISO depend on the structure of the input matrix A .

Symmetric Matrices

The solver first computes a symmetric fill-in reducing permutation P based on either the minimum degree algorithm [Liu85] or the nested dissection algorithm from the METIS package [Karypis98] (both included with Intel MKL), followed by the parallel left-right looking numerical Cholesky factorization [Schenk00-2] of $PAP^T = LL^T$ for symmetric positive-definite matrices, or $PAP^T = LDL^T$ for symmetric indefinite matrices. The solver uses diagonal pivoting, or 1x1 and 2x2

Bunch-Kaufman pivoting for symmetric indefinite matrices. An approximation of X is found by forward and backward substitution and optional iterative refinement.

Whenever numerically acceptable 1×1 and 2×2 pivots cannot be found within the diagonal supernode block, the coefficient matrix is perturbed. One or two passes of iterative refinement may be required to correct the effect of the perturbations. This restricting notion of pivoting with iterative refinement is effective for highly indefinite symmetric systems. Furthermore, for a large set of matrices from different applications areas, this method is as accurate as a direct factorization method that uses complete sparse pivoting techniques [Schenk04].

Another method of improving the pivoting accuracy is to use symmetric weighted matching algorithms. These algorithms identify large entries in the coefficient matrix A that, if permuted close to the diagonal, permit the factorization process to identify more acceptable pivots and proceed with fewer pivot perturbations. These algorithms are based on maximum weighted matchings and improve the quality of the factor in a complementary way to the alternative of using more complete pivoting techniques.

The inertia is also computed for real symmetric indefinite matrices.

Structurally Symmetric Matrices

The solver first computes a symmetric fill-in reducing permutation P followed by the parallel numerical factorization of $PAP^T = QLU^T$. The solver uses partial pivoting in the supernodes and an approximation of X is found by forward and backward substitution and optional iterative refinement.

Nonsymmetric Matrices

The solver first computes a nonsymmetric permutation P_{MPS} and scaling matrices D_r and D_c with the aim of placing large entries on the diagonal to enhance reliability of the numerical factorization process [Duff99]. In the next step the solver computes a fill-in reducing permutation P based on the matrix $P_{MPS}A + (P_{MPS}A)^T$ followed by the parallel numerical factorization

$$QLUR = PP_{MPS}D_rAD_cP$$

with supernode pivoting matrices Q and R . When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99]. The magnitude of the potential pivot is tested against a constant threshold of

$$\text{alpha} = \text{eps} * \|A2\|_{\text{inf}},$$

where eps is the machine precision, $A2 = P * P_{MPS} * D_r * A * D_c * P$, and $\|A2\|_{\text{inf}}$ is the infinity norm of A . Any tiny pivots encountered during elimination are set to the sign $(1_{II}) * \text{eps} * \|A2\|_{\text{inf}}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Although many failures could render the factorization well-defined but essentially useless, in practice the diagonal elements are rarely modified for a large class of matrices. The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed.

Sparse Data Storage

Sparse data storage in Intel MKL PARDISO follows the 3-array variation of the compressed sparse row (CSR3) format described in [Sparse Matrix Storage Format](#) with the Intel MKL PARDISO parameters *ja* standing for *columns*, *ia* for *rowIndex*, and *a* for *values*. The algorithms in Intel MKL PARDISO require column indices *ja* to be in increasing order per row and that the diagonal element in each row be present for any structurally symmetric matrix. For symmetric or nonsymmetric matrices the diagonal elements which are equal to zero are not necessary.

CAUTION

Intel MKL PARDISO column indices ja must be in increasing order per row. You can validate the sparse matrix structure with the matrix checker ([iparm\(27\)](#))

NOTE

While the presence of zero diagonal elements for symmetric matrices is not required, you should explicitly set zero diagonal elements for symmetric matrices. Otherwise, Intel MKL PARDISO creates internal copies of arrays ia , ja , and a full of diagonal elements, which require additional memory and computational time. However, the memory and time required the diagonal elements in internal arrays is usually not significant compared to the memory and the time required to factor and solve the matrix.

Storage of Matrices

By default, Intel MKL PARDISO stores matrices in RAM. However, you can specify that Intel MKL PARDISO store matrices on disk by setting [iparm\(60\)](#). This is referred to as in-core (IC) and out-of-core (OOC), respectively.

OOC parameters can be set in a configuration file. You can set the path to this file and its name using environmental variables `MKL_PARDISO_OOC_CFG_PATH` and `MKL_PARDISO_OOC_CFG_FILE_NAME`.

These variables specify the path and filename as follows:

`<MKL_PARDISO_OOC_CFG_PATH>/<MKL_PARDISO_OOC_CFG_FILE_NAME>` for Linux* OS and OS X*, and
`<MKL_PARDISO_OOC_CFG_PATH>\<MKL_PARDISO_OOC_CFG_FILE_NAME>` for Windows* OS.

By default, the name of the file is `pardiso_ooc.cfg` and it is placed in the current directory.

All temporary data files can be deleted or stored when the calculations are completed in accordance with the value of the environmental variable `MKL_PARDISO_OOC_KEEP_FILE`. If it is not set or if it is set to 1, then all files are deleted. If it is set to 0, then all files are stored.

By default, the OOC version of Intel MKL PARDISO uses the current directory for storing data, and all work arrays associated with the matrix factors are stored in files named `ooc_temp` with different extensions. These default values can be changed by using the environmental variable `MKL_PARDISO_OOC_PATH`.

To set the environmental variables `MKL_PARDISO_OOC_MAX_CORE_SIZE`, `MKL_PARDISO_OOC_MAX_SWAP_SIZE`, `MKL_PARDISO_OOC_KEEP_FILE`, and `MKL_PARDISO_OOC_PATH`, create the configuration file with the following lines:

```
MKL_PARDISO_OOC_PATH = <path>\ooc_file
MKL_PARDISO_OOC_MAX_CORE_SIZE = N
MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

where `<path>` is the directory for storing data, `ooc_file` is the file name without any extension, N is the maximum size of RAM in megabytes available for Intel MKL PARDISO (default value is 2000 MB), K is the maximum swap size in megabytes available for Intel MKL PARDISO (default value is 0 MB). Do not set $N+K$ greater than the size of the RAM plus the size of the swap. Be sure to allow enough free memory for the operating system and any other processes which are necessary.

CAUTION

The maximum length of the path lines in the configuration files is 1000 characters.

Alternatively the environment variables can be set via command line.

For Linux* OS and OS X*:

```
export MKL_PARDISO_OOC_PATH = <path>/ooc_file
export MKL_PARDISO_OOC_MAX_CORE_SIZE = N
export MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
export MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

For Windows* OS:

```
set MKL_PARDISO_OOC_PATH = <path>\ooc_file
set MKL_PARDISO_OOC_MAX_CORE_SIZE = N
set MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
set MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

NOTE

The values specified in a command line have higher priorities: if a variable is changed in the configuration file and in the command line, OOC version of Intel MKL PARDISO uses only the value defined in the command line.

Direct-Iterative Preconditioning for Nonsymmetric Linear Systems

The solver uses a combination of direct and iterative methods [Sonn89] to accelerate the linear solution process for transient simulation. Most applications of sparse solvers require solutions of systems with gradually changing values of the nonzero coefficient matrix, but with an identical sparsity pattern. In these applications, the analysis phase of the solvers has to be performed only once and the numerical factorizations are the important time-consuming steps during the simulation. Intel MKL PARDISO uses a numerical factorization and applies the factors in a preconditioned Krylow-Subspace iteration. If the iteration does not converge, the solver automatically switches back to the numerical factorization. This method can be applied to nonsymmetric matrices in Intel MKL PARDISO. You can select the method using the *iparm*(4) input parameter. The *iparm*(20) parameter returns the error status after running Intel MKL PARDISO.

Single and Double Precision Computations

Intel MKL PARDISO solves tasks using single or double precision. Each precision has its benefits and drawbacks. Double precision variables have more digits to store value, so the solver uses more memory for keeping data. But this mode solves matrices with better accuracy, which is especially important for input matrices with large condition numbers.

Single precision variables have fewer digits to store values, so the solver uses less memory than in the double precision mode. Additionally this mode usually takes less time. But as computations are made less precisely, only some systems of equations can be solved accurately enough using single precision.

Separate Forward and Backward Substitution

The solver execution step (see *parameter phase* = 33 below) can be divided into two or three separate substitutions: forward, backward, and possible diagonal. This separation can be explained by the examples of solving systems with different matrix types.

A real symmetric positive definite matrix A (*mtype* = 2) is factored by Intel MKL PARDISO as $A = L^T L$. In this case the solution of the system $A^T x = b$ can be found as sequence of substitutions: $L^T y = b$ (forward substitution, *phase* = 331) and $L^T x = y$ (backward substitution, *phase* = 333).

A real nonsymmetric matrix A (*mtype* = 11) is factored by Intel MKL PARDISO as $A = L^T U$. In this case the solution of the system $A^T x = b$ can be found by the following sequence: $L^T y = b$ (forward substitution, *phase* = 331) and $U^T x = y$ (backward substitution, *phase* = 333).

Solving a system with a real symmetric indefinite matrix A (*mtype* = -2) is slightly different from the cases above. Intel MKL PARDISO factors this matrix as $A = LDL^T$, and the solution of the system $A^T x = b$ can be calculated as the following sequence of substitutions: $L^T y = b$ (forward substitution, *phase* = 331), $D^T v = y$

(diagonal substitution, $\text{phase} = 332$), and finally $L^T x = v$ (backward substitution, $\text{phase} = 333$). Diagonal substitution makes sense only for symmetric indefinite matrices ($mtype = -2, -4, 6$). For matrices of other types a solution can be found as described in the first two examples.

CAUTION

The number of refinement steps ([iparm\(8\)](#)) must be set to zero if a solution is calculated with separate substitutions ($\text{phase} = 331, 332, 333$), otherwise Intel MKL PARDISO produces the wrong result.

NOTE

Different pivoting ([iparm\(21\)](#)) produces different LDL^T factorization. Therefore results of forward, diagonal and backward substitutions with diagonal pivoting can differ from results of the same steps with Bunch-Kaufman pivoting. Of course, the final results of sequential execution of forward, diagonal and backward substitution are equal to the results of the full solving step ($\text{phase}=33$) regardless of the pivoting used.

Callback Function for Pivoting Control

In-core Intel MKL PARDISO allows you to control pivoting with a callback routine, [mkl_pardiso_pivot](#). You can then use the [pardiso_getdiag](#) routine to access the diagonal elements. Set [iparm\(56\)](#) to 1 in order to use the callback functionality.

[pardiso](#)

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax

Fortran:

```
call pardiso (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs, iparm, msglvl,
b, x, error)
```

C:

```
void pardiso (_MKL_DSS_HANDLE_t pt, MKL_INT *maxfct, MKL_INT *mnum, MKL_INT *mtype,
MKL_INT *phase, MKL_INT *n, void *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *perm, MKL_INT
*nrhs, MKL_INT *iparm, MKL_INT *msglvl, void *b, void *x, MKL_INT *error);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_pardiso.f90`
- C: `mkl.h`

Description

The routine `pardiso` calculates the solution of a set of sparse linear equations

$A \cdot X = B$

with single or multiple right-hand sides, using a parallel LU , LDL , or LL^T factorization, where A is an n -by- n matrix, and X and B are n -by- $nrhs$ vectors or matrices.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters**NOTE**

The types given for parameters in this section are specified in FORTRAN 77 notation. See [Intel MKL PARDISO Parameters in Tabular Form](#) for detailed description of types of Intel MKL PARDISO parameters in Fortran 90 and C notations.

<i>pt</i>	INTEGER for 32-bit or 64-bit architectures INTEGER*8 for 64-bit architectures Array with dimension of 64. Handle to internal data structure. The entries must be set to zero prior to the first call to <code>pardiso</code> . Unique for factorization.
-----------	---

CAUTION

After the first call to `pardiso` do not directly modify *pt*, as that could cause a serious memory leak.

Use the `pardiso_handle_store` or `pardiso_handle_store_64` routine to store the content of *pt* to a file. Restore the contents of *pt* from the file using `pardiso_handle_restore` or `pardiso_handle_restore_64`. Use `pardiso_handle_store` and `pardiso_handle_restore` with `pardiso`, and `pardiso_handle_store_64` and `pardiso_handle_restore_64` with `pardiso_64`.

<i>maxfct</i>	INTEGER Maximum number of factors with identical sparsity structure that must be kept in memory at the same time. In most applications this value is equal to 1. It is possible to store several different factorizations with the same nonzero structure at the same time in the internal data structure management of the solver. <code>pardiso</code> can process several matrices with an identical matrix sparsity pattern and it can store the factors of these matrices at the same time. Matrices with a different sparsity structure can be kept in memory with different memory address pointers <i>pt</i> .
---------------	--

<i>mnum</i>	INTEGER Indicates the actual matrix for the solution phase. With this scalar you can define which matrix to factorize. The value must be: $1 \leq mnum \leq maxfct$. In most applications this value is 1.
-------------	---

<i>mtype</i>	INTEGER Defines the matrix type, which influences the pivoting method. The Intel MKL PARDISO solver supports the following matrices:
--------------	---

1	real and structurally symmetric
2	real and symmetric positive definite
-2	real and symmetric indefinite
3	complex and structurally symmetric
4	complex and Hermitian positive definite
-4	complex and Hermitian indefinite
6	complex and symmetric
11	real and nonsymmetric
13	complex and nonsymmetric

phase

INTEGER

Controls the execution of the solver. Usually it is a two- or three-digit integer. The first digit indicates the starting phase of execution and the second digit indicates the ending phase. Intel MKL PARDISO has the following phases of execution:

- Phase 1: Fill-reduction analysis and symbolic factorization
- Phase 2: Numerical factorization
- Phase 3: Forward and Backward solve including optional iterative refinement

This phase can be divided into two or three separate substitutions: forward, backward, and diagonal (see [Separate Forward and Backward Substitution](#)).

- Memory release phase (*phase*= 0 or *phase*= -1)

If a previous call to the routine has computed information from previous phases, execution may start at any phase. The *phase* parameter can have the following values:

<i>phase</i>	Solver Execution Steps
11	Analysis
12	Analysis, numerical factorization
13	Analysis, numerical factorization, solve, iterative refinement
22	Numerical factorization
23	Numerical factorization, solve, iterative refinement
33	Solve, iterative refinement
331	like <i>phase</i> =33, but only forward substitution
332	like <i>phase</i> =33, but only diagonal substitution (if available)

<i>phase</i>	Solver Execution Steps
333	like <i>phase</i> =33, but only backward substitution
0	Release internal memory for <i>L</i> and <i>U</i> matrix number <i>mnum</i>
-1	Release all internal memory for all matrices

If *iparm*(36) = 0, phases 331, 332, and 333 perform this decomposition:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{12} & L_{22} \end{bmatrix} \begin{bmatrix} D_{11} & 0 \\ 0 & D_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{21} \\ 0 & U_{22} \end{bmatrix}$$

If *iparm*(36) = 2, phases 331, 332, and 333 perform a different decomposition:

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{12} & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} U_{11} & U_{21} \\ 0 & I \end{bmatrix}$$

You can supply a custom implementation for phase 332 instead of calling *pardiso*. For example, it can be implemented with dense LAPACK functionality. Custom implementation also allows you to substitute the matrix *S* with your own.

NOTE

For very large Schur complement matrices use LAPACK functionality to compute the Schur complement vector instead of the Intel MKL PARDISO phase 332 implementation.

n

INTEGER

Number of equations in the sparse linear systems of equations *A***X* = *B*.
Constraint: *n* > 0.

a

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Intel MKL PARDISO (*iparm*(28)=0)

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Intel MKL PARDISO (*iparm*(28)=1)

DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Intel MKL PARDISO (*iparm*(28)=0)

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Intel MKL PARDISO (*iparm*(28)=1)

Array. Contains the non-zero elements of the coefficient matrix *A* corresponding to the indices in *ja*. The size of *a* is the same as that of *ja* and the coefficient matrix can be either real or complex. The matrix must be stored in the three-array variant of the compressed sparse row (CSR3) format with increasing values of *ja* for each row. Refer to *values* array description in [Storage Formats for the Direct Sparse Solvers](#) for more details.

ia

INTEGER

Array, dimension $(n+1)$. For $i \leq n$, $ia(i)$ points to the first column index of row i in the array ja in compressed sparse row format. That is, $ia(i)$ gives the index of the element in array a that contains the first non-zero element from row i of A . The last element $ia(n+1)$ is taken to be equal to the number of non-zero elements in A , plus one. Refer to `rowIndex` array description in [Storage Formats for the Direct Sparse Solvers](#) for more details. The array ia is also accessed in all phases of the solution process.

Indexing of ia is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter [`iparm\(35\)`](#).

ja

INTEGER

Array $ja(*)$ contains column indices of the sparse matrix A stored in compressed sparse row format. It is important that the indices are in increasing order per row. The array ja is also accessed in all phases of the solution process. For structurally symmetric matrices it is assumed that all diagonal elements are stored (even if they are zeros) in the list of non-zero elements in a and ja . For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for `columns` array in [Storage Formats for the Direct Sparse Solvers](#).

Indexing of ja is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter [`iparm\(35\)`](#).

perm

INTEGER

Array, dimension (n) . Depending on the value of [`iparm\(5\)`](#) and [`iparm\(31\)`](#), either holds the permutation vector of size n or specifies elements used for computing a partial solution.

- If $iparm(5) = 1$, $iparm(31) = 0$, and $iparm(36) = 0$, $perm$ specifies the fill-in reducing ordering to the solver. Let A be the original matrix and $C = P^* A^* P^T$ be the permuted matrix. Row (column) i of C is the $perm(i)$ row (column) of A . The array $perm$ is also used to return the permutation vector calculated during fill-in reducing ordering stage.

NOTE

Be aware that setting $iparm(5) = 1$ prevents use of a parallel algorithm for the solve step.

- If $iparm(5) = 2$, $iparm(31) = 0$, and $iparm(36) = 0$, the computed permutation vector is returned in the $perm$ array.
- If $iparm(5) = 0$, $iparm(31) > 0$, and $iparm(36) = 0$, $perm$ specifies elements of the right-hand side to use or of the solution to compute for a partial solution.
- If $iparm(5) = 0$, $iparm(31) = 0$, and $iparm(36) > 0$, $perm$ specifies elements for a Schur complement.

See [`iparm\(5\)`](#) and [`iparm\(31\)`](#) for more details.

Indexing of $perm$ is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter [`iparm\(35\)`](#).

nrhs

INTEGER

Number of right-hand sides that need to be solved for.

iparm

INTEGER

Array, dimension (64). This array is used to pass various parameters to Intel MKL PARDISO and to return some useful information after execution of the solver.

See [pardiso iparm Parameter](#) for more details about the *iparm* parameters.

msglvl

INTEGER

Message level information. If *msglvl* = 0 then pardiso generates no output, if *msglvl* = 1 the solver prints statistical information to the screen.

b

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Intel MKL PARDISO (*iparm*(28)=0)

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Intel MKL PARDISO (*iparm*(28)=1)

DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Intel MKL PARDISO (*iparm*(28)=0)

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Intel MKL PARDISO (*iparm*(28)=1)

Array, dimension (*n*, *nrhs*). On entry, contains the right-hand side vector/ matrix *B*, which is placed in memory contiguously. The *b*(*i*+(*k*-1)×*nrhs*) must hold the *i*-th component of *k*-th right-hand side vector. Note that *b* is only accessed in the solution phase.

Output Parameters

(See also [Intel MKL PARDISO Parameters in Tabular Form](#).)

pt

Handle to internal data structure.

iparm

On output, some *iparm* values report information such as the numbers of non-zero elements in the factors.

See [pardiso iparm Parameter](#) for more details about the *iparm* parameters.

b

On output, the array is replaced with the solution if *iparm*(6) = 1.

x

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Intel MKL PARDISO (*iparm*(28)=0)

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Intel MKL PARDISO (*iparm*(28)=1)

DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Intel MKL PARDISO (*iparm*(28)=0)

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Intel MKL PARDISO (*iparm*(28)=1)

Array, dimension ($n, nrhs$). If `iparm(6) = 0` it contains solution vector/matrix X , which is placed contiguously in memory. The $x(i + (k-1) \times n)$ element must hold the i -th component of the k -th solution vector. Note that x is only accessed in the solution phase.

`error`

INTEGER

The error indicator according to the below table:

<code>error</code>	Information
0	no error
-1	input inconsistent
-2	not enough memory
-3	reordering problem
-4	zero pivot, numerical factorization or iterative refinement problem
-5	unclassified (internal) error
-6	reordering failed (matrix types 11 and 13 only)
-7	diagonal matrix is singular
-8	32-bit integer overflow problem
-9	not enough memory for OOC
-10	error opening OOC files
-11	read/write error with OOC files
-12	(<code>pardiso_64</code> only) <code>pardiso_64</code> called from 32-bit library

pardisoinit

Initialize Intel MKL PARDISO with default parameters in accordance with the matrix type.

Syntax

Fortran:

```
call pardisoinit (pt, mtype, iparm)
```

C:

```
void pardisoinit (_MKL_DSS_HANDLE_t pt, MKL_INT *mtype, MKL_INT *iparm);
```

Include Files

- **Fortran:** `mkl.fi`
- **Fortran 90:** `mkl_pardiso.f90`
- **C:** `mkl.h`

Description

This function initializes Intel MKL PARDISO internal address pointer *pt* with zero values (as needed for the very first call of *pardiso*) and sets default *iparm* values in accordance with the matrix type. Intel MKL supplies the *pardisoinit* routine to be compatible with PARDISO 3.2 or lower.

NOTE

An alternative way to set default *iparm* values is to call *pardiso* in the analysis phase with *iparm(1)=0*. In this case you must initialize the internal address pointer *pt* with zero values manually.

NOTE

The *pardisoinit* routine initializes only the in-core version of Intel MKL PARDISO. Switching on the out-of core version of Intel MKL PARDISO as well as changing default *iparm* values can be done after the call to *pardisoinit* but before the first call to *pardiso*.

Input Parameters

NOTE

Parameters types in this section are specified in FORTRAN 77 notation. See [Intel MKL PARDISO Parameters in Tabular Form](#) section for detailed description of types of Intel MKL PARDISO parameters in C/ Fortran 90 notations.

mtype

INTEGER

This scalar value defines the matrix type. Based on this value *pardisoinit* sets default values for the *iparm* array. Refer to the section [Intel MKL PARDISO Parameters in Tabular Form](#) for more details about the default values of Intel MKL PARDISO

Output Parameters

pt

INTEGER for 32-bit or 64-bit architectures

INTEGER*8 for 64-bit architectures

Array with a dimension of 64. Solver internal data address pointer. These addresses are passed to the solver, and all related internal memory management is organized through this array. The *pardisoinit* routine nullifies the array *pt*.

NOTE

It is very important that the pointer *pt* is initialized with zero before the first call of Intel MKL PARDISO. After that first call you should never modify the pointer, because it could cause a serious memory leak or a crash.

iparm

INTEGER

Array with a dimension of 64. This array is used to pass various parameters to Intel MKL PARDISO and to return some useful information after execution of the solver. The pardisoinit routine fills-in the *iparm* array with the default values. Refer to the section [Intel MKL PARDISO Parameters in Tabular Form](#) for more details about the default values of Intel MKL PARDISO.

[pardiso_64](#)

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides, 64-bit integer version.

Syntax

Fortran:

```
call pardiso_64 (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs, iparm,
msglvl, b, x, error)
```

C:

```
void pardiso_64 (_MKL_DSS_HANDLE_t pt, long long int *maxfct, long long int *mnum, long
long int *mtype, long long int *phase, long long int *n, void *a, long long int *ia,
long long int *ja, long long int *perm, long long int *nrhs, long long int *iparm, long
long int *msglvl, void *b, void *x, long long int *error);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_pardiso.f90
- C: mkl.h

Description

`pardiso_64` is an alternative ILP64 (64-bit integer) version of the `pardiso` routine (see [Description](#) section for more details). The interface of `pardiso_64` is the same as the interface of `pardiso`, but it accepts and returns all INTEGER data as INTEGER*8.

Use `pardiso_64` when `pardiso` for solving large matrices (with the number of non-zero elements on the order of 500 million or more). You can use it together with the usual LP64 interfaces for the rest of Intel MKL functionality. In other words, if you use 64-bit integer version (`pardiso_64`), you do not need to re-link your applications with ILP64 libraries. Take into account that `pardiso_64` may perform slower than regular `pardiso` on the reordering and symbolic factorization phase.

NOTE

`pardiso_64` is supported only in the 64-bit libraries. If `pardiso_64` is called from the 32-bit libraries, it returns `error == -12`.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

The input parameters of `pardiso_64` are the same as the [input parameters of `pardiso`](#), but `pardiso_64` accepts all INTEGER data as `INTEGER*8`.

Output Parameters

The output parameters of `pardiso_64` are the same as the [output parameters of `pardiso`](#), but `pardiso_64` returns all INTEGER data as `INTEGER*8`.

`pardiso_getenv`, `pardiso_setenv`

Retrieves additional values from or sets additional values in the Intel MKL PARDISO handle.

Syntax

Fortran:

```
error = pardiso_getenv(handle, param, value)
error = pardiso_setenv(handle, param, value)
```

C:

```
MKL_INT pardiso_getenv (const _MKL_DSS_HANDLE_t handle, const enum PARDISO_ENV_PARAM
*param, char *value);

MKL_INT pardiso_setenv (_MKL_DSS_HANDLE_t handle , const enum PARDISO_ENV_PARAM *param,
const char *value);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_pardiso.f90`
- C: `mkl.h`

NOTE

`pardiso_setenv` requires the `value` parameter to be converted to the string in C notation if it is called from Fortran. You can do this using `mkl_cvt_to_null_terminated_str` subroutine declared in the `mkl_dss.f77` or `mkl_dss.f90` include files (see the example below).

Description

These functions operate with the Intel MKL PARDISO handle. The `pardiso_getenv` routine retrieves additional values from the Intel MKL PARDISO handle, and `pardiso_setenv` sets specified values in the Intel MKL PARDISO handle.

These functions enable retrieving and setting the name of the Intel MKL PARDISO OOC file.

To retrieve the Intel MKL PARDISO OOC file name, you can apply this function to any properly-created handle.

To set the the Intel MKL PARDISO OOC file name in the handle you must call the function before the reordering stage. This is because the OOC file name is stored in the handle after the reordering stage and it is not changed during further computations.

NOTE

A 1024-byte internal buffer is used inside Intel MKL PARDISO for storing the OOC file name. Allocate a 1024-byte buffer for passing to the `pardiso_getenv` function as the `value` parameter.

Input Parameters

<i>handle</i>	Fortran: MKL_FORTRAN_HANDLE Intel MKL PARDISO handle for which to set and from which to retrieve information. (See DSS Interface Description for structure description)
<i>param</i>	Fortran: INTEGER. Specifies the required parameter. The only value is <code>PARDISO_OOC_FILE_NAME</code> , defined in the corresponding include file.
<i>value</i>	Fortran: CHARACTER array. Input parameter for <code>pardiso_setenv</code> . Contains the name of the OOC file that must be used in the handle.

Output Parameters

<i>handle</i>	Output parameter for <code>pardiso_setenv</code> . Data object of the MKL_DSS_HANDLE type (see DSS Interface Description).
<i>value</i>	Output parameter for <code>pardiso_getenv</code> . Contains the name of the OOC file that must be used in the handle.

Example (FORTRAN 90)

```

INCLUDE 'mkl_pardiso.f90'
INCLUDE 'mkl_dss.f90'
PROGRAM pardiso_sym_f90
USE mkl_pardiso
USE mkl_dss

INTEGER*8 pt(64)
CHARACTER*1024 file_name
INTEGER buff(256), bufLen, error

pt(1:64) = 0

file_name = 'pardiso_ooc_file'
bufLen = len_trim(file_name)
call mkl_cvt_to_null_terminated_str(buff, bufLen, trim(file_name))
error = pardiso_setenv(pt, PARDISO_OOC_FILE_NAME, buff)

! call pardiso() here

END PROGRAM

```

mkl_pardiso_pivot

Replaces routine which handles Intel MKL PARDISO pivots with user-defined routine.

Syntax

Fortran:

```
call mkl_pardiso_pivot (ai, bi, eps)
```

C:

```
void mkl_pardiso_pivot (void *ai, void *bi, void *eps);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_pardiso.f90
- C: mkl.h

Description

The `mkl_pardiso_pivot` routine allows you to handle diagonal elements which arise during numerical factorization that are zero or near zero. By default, Intel MKL PARDISO determines that a diagonal element `bi` is a pivot if `bi < eps`, and if so, replaces it with `eps`. But you can provide your own routine to modify the resulting factorized matrix in case there are small elements on the diagonal during the factorization step.

NOTE

To use this routine, you must set `iparm(56)` to 1 before the main `pardiso` loop.

NOTE

This routine is only available for in-core Intel MKL PARDISO.

Input Parameters

<code>ai</code>	DOUBLE PRECISION - for real types of matrices (<code>mtype=2, -2, 4, and 6</code>) and for double precision Intel MKL PARDISO (<code>iparm(28)=0</code>)
	Diagonal element of initial matrix corresponding to pivot element.
<code>bi</code>	DOUBLE PRECISION - for real types of matrices (<code>mtype=2, -2, 4, and 6</code>) and for double precision Intel MKL PARDISO (<code>iparm(28)=0</code>)
	Diagonal element of factorized matrix that could be chosen as a pivot element.
<code>eps</code>	DOUBLE PRECISION
	Scalar to compare with diagonal of factorized matrix. On input equal to parameter described by <code>iparm(10)</code> .

Output Parameters

<code>bi</code>	In case element is chosen as a pivot, value with which to replace the pivot.
-----------------	--

`pardiso_getdiag`

Returns diagonal elements of initial and factorized matrix.

Syntax

Fortran:

```
call pardiso_getdiag (pt, df, da, mnum, error)
```

C:

```
void pardiso_getdiag (_MKL_DSS_HANDLE_t pt, void *df, void *da, MKL_INT *mnum,
MKL_INT *error);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_pardiso.f90
- C: mkl.h

Description

This routine returns the diagonal elements of the initial and factorized matrix for a real or Hermitian matrix.

NOTE

In order to use this routine, you must set [iparm\(56\)](#) to 1 before the main pardiso loop.

Input Parameters

<i>pt</i>	INTEGER for 32-bit or 64-bit architectures INTEGER*8 for 64-bit architectures Array with a dimension of 64. Handle to internal data structure for the Intel MKL PARDISO solver. The entries must be set to zero prior to the first call to pardiso. Unique for factorization.
<i>mnum</i>	INTEGER Indicates the actual matrix for the solution phase of the Intel MKL PARDISO solver. With this scalar you can define the diagonal elements of the factorized matrix that you want to obtain. The value must be: $1 \leq mnum \leq maxfct$. In most applications this value is 1.

Output Parameters

<i>df</i>	DOUBLE PRECISION - for real types of matrices and for double precision Intel MKL PARDISO (iparm(28)=0) REAL - for real types of matrices and for single precision Intel MKL PARDISO (iparm(28)=1) DOUBLE COMPLEX - for complex types of matrices and for double precision Intel MKL PARDISO (iparm(28)=0) COMPLEX - for complex types of matrices and for single precision Intel MKL PARDISO (iparm(28)=1) Array with a dimension of <i>n</i> . Contains diagonal elements of the factorized matrix after factorization.
<i>da</i>	DOUBLE PRECISION - for real types of matrices and for double precision Intel MKL PARDISO (iparm(28)=0)

REAL - for real types of matrices and for single precision Intel MKL PARDISO (*iparm*(28)=1)

DOUBLE COMPLEX - for complex types of matrices and for double precision Intel MKL PARDISO (*iparm*(28)=0)

COMPLEX - for complex types of matrices and for single precision Intel MKL PARDISO (*iparm*(28)=1)

Array with a dimension of *n*. Contains diagonal elements of the initial matrix.

error

INTEGER

The error indicator.

error	Information
0	no error
-1	Diagonal information not turned on before pardiso main loop (<i>iparm</i> (56)=0).

[pardiso_handle_store](#)

Store internal structures from pardiso to a file.

Syntax

Fortran:

```
call pardiso_handle_store (pt, dirname, error)
```

C:

```
void pardiso_handle_store (_MKL_DSS_HANDLE_t pt, char *dirname, MKL_INT *error);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_pardiso.f90
- C: mkl.h

Description

This function stores Intel MKL PARDISO structures to a file, allowing you to store Intel MKL PARDISO internal structures between the stages of the `pardiso` routine. The `pardiso_handle_restore` routine can restore the Intel MKL PARDISO internal structures from the file.

Input Parameters

pt

INTEGER for 32-bit or 64-bit architectures

INTEGER*8 for 64-bit architectures

Array with a dimension of 64. Handle to internal data structure.

dirname

CHARACTER

String containing the name of the directory to which to write the files with the content of the internal structures. Use an empty string ("") to specify the current directory. The routine creates a file named `handle.pds` in the directory.

Output Parameters

`error` INTEGER

The error indicator.

<code>error</code>	Information
0	No error.
-2	Not enough memory.
-10	Cannot open file for writing.
-11	Error while writing to file.
-13	Wrong file format.

pardiso_handle_restore

Restore pardiso internal structures from a file.

Syntax

Fortran:

```
call pardiso_handle_restore (pt, dirname, error)
```

C:

```
void pardiso_handle_restore (_MKL_DSS_HANDLE_t pt, char *dirname, MKL_INT *error);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_pardiso.f90`
- C: `mkl.h`

Description

This function restores Intel MKL PARDISO structures from a file. This allows you to restore Intel MKL PARDISO internal structures stored by `pardiso_handle_store` after a phase of the `pardiso` routine and continue execution of the next phase.

Input Parameters

`dirname` CHARACTER

String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Output Parameters

pt INTEGER for 32-bit or 64-bit architectures
 INTEGER*8 for 64-bit architectures
 Array with a dimension of 64. Handle to internal data structure.

error INTEGER

The error indicator.

error	Information
0	No error.
-2	Not enough memory.
-10	Cannot open file for reading.
-11	Error while reading from file.
-13	Wrong file format.

pardiso_handle_delete

Delete files with pardiso internal structure data.

Syntax

Fortran:

```
call pardiso_handle_delete (dirname, error)
```

C:

```
void pardiso_handle_delete (char *dirname, MKL_INT *error);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_pardiso.f90
- C: mkl.h

Description

This function deletes files generated with `pardiso_handle_store` that contain Intel MKL PARDISO internal structures.

Input Parameters

dirname CHARACTER

String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Output Parameters

error INTEGER

The error indicator.

error	Information
0	No error.
-10	Cannot delete files.

[pardiso_handle_store_64](#)

Store internal structures from pardiso_64 to a file.

Syntax

Fortran:

```
call pardiso_handle_store_64 (pt, dirname, error)
```

C:

```
void pardiso_handle_store_64 (_MKL_DSS_HANDLE_t pt, char *dirname, MKL_INT *error);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_pardiso.f90
- C: mkl.h

Description

This function stores Intel MKL PARDISO structures to a file, allowing you to store Intel MKL PARDISO internal structures between the stages of the [pardiso_64](#) routine. The [pardiso_handle_restore_64](#) routine can restore the Intel MKL PARDISO internal structures from the file.

Input Parameters

pt INTEGER*8 for 64-bit architectures
 Array with a dimension of 64. Handle to internal data structure.

dirname CHARACTER
 String containing the name of the directory to which to write the files with the content of the internal structures. Use an empty string ("") to specify the current directory. The routine creates a file named handle.pds in the directory.

Output Parameters

error INTEGER
 The error indicator.

error	Information
0	No error.
-2	Not enough memory.
-10	Cannot open file for writing.

error	Information
-11	Error while writing to file.
-12	Not supported in 32-bit library - routine is only supported in 64-bit libraries.
-13	Wrong file format.

pardiso_handle_restore_64

Restore pardiso_64 internal structures from a file.

Syntax

Fortran:

```
call pardiso_handle_restore_64 (pt, dirname, error)
```

C:

```
void pardiso_handle_restore_64 (_MKL_DSS_HANDLE_t pt, char *dirname, MKL_INT *error);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_pardiso.f90
- C: mkl.h

Description

This function restores Intel MKL PARDISO structures from a file. This allows you to restore Intel MKL PARDISO internal structures stored by `pardiso_handle_store_64` after a phase of the `pardiso_64` routine and continue execution of the next phase.

Input Parameters

<i>dirname</i>	CHARACTER
	String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Input Parameters

<i>pt</i>	INTEGER for 32-bit or 64-bit architectures INTEGER*8 for 64-bit architectures
	Array with a dimension of 64. Handle to internal data structure.

error INTEGER

The error indicator.

error Information

0	No error.
-2	Not enough memory.

error	Information
-10	Cannot open file for reading.
-11	Error while reading from file.
-13	Wrong file format.

pardiso_handle_delete_64

Syntax

Delete files with `pardiso_64` internal structure data.

Fortran:

```
call pardiso_handle_delete_64 (dirname, error)
```

C:

```
void pardiso_handle_delete_64 (char *dirname, MKL_INT *error);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_pardiso.f90`
- C: `mkl.h`

Description

This function deletes files generated with `pardiso_handle_dump_64` that contain Intel MKL PARDISO internal structures.

Input Parameters

<i>dirname</i>	CHARACTER
	String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

Output Parameters

error	Information
0	No error.
-10	Cannot delete files.
-12	Not supported in 32-bit library - routine is only supported in 64-bit libraries.

Intel MKL PARDISO Parameters in Tabular Form

The following table lists all parameters of Intel MKL PARDISO and gives their brief descriptions.

Parameter	Type	Description	Values	Comments	In/ Out
<i>pt(64)</i>	FORTRAN 77: INTEGER on 32-bit architectures, INTEGER*8 on 64- bit architectures Fortran 90: TYPE (MKL_PARDISO_HANDLE) , INTENT (INOUT) C: void*	Solver internal data address pointer	0	Must be initialized with zeros and never be modified later	in/ out
<i>maxfct</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: MKL_INT*	Maximal number of factors in memory	>0	Generally used value is 1	in
<i>mnum</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: MKL_INT*	The number of matrix (from 1 to <i>maxfct</i>) to solve	[1: <i>maxfct</i>]	Generally used value is 1	in
<i>mtype</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: MKL_INT*	Matrix type	1 2 -2 3 4 -4 6 11 13	Real and structurally symmetric Real and symmetric positive definite Real and symmetric indefinite Complex and structurally symmetric Complex and Hermitian positive definite Complex and Hermitian indefinite Complex and symmetric matrix Real and nonsymmetric matrix Complex and nonsymmetric matrix	in
<i>phase</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: MKL_INT*	Controls the execution of the solver For <i>iparm(36) ></i> 0, phases 331, 332, and 333	11 12 13	Analysis Analysis, numerical factorization Analysis, numerical factorization, solve	in

Parameter	Type	Description	Values	Comments	In/ Out
		perform a different decomposition. See the <i>phase</i> parameter of pardiso for details.	22 23 33 331 332 333 0 -1	Numerical factorization Numerical factorization, solve Solve, iterative refinement <i>phase</i> =33, but only forward substitution <i>phase</i> =33, but only diagonal substitution <i>phase</i> =33, but only backward substitution Release internal memory for L and U of the matrix number <i>mnum</i> Release all internal memory for all matrices	
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: MKL_INT*	Number of equations in the sparse linear system $A*X = B$	>0		in
<i>a (*)</i>	FORTRAN 77: PARDISO_DATA_TYPE 1) Fortran 90: PARDISO_DATA_TYPE 1), INTENT(IN) C: void*	Contains the non-zero elements of the coefficient matrix <i>A</i>	*	The size of <i>a</i> is the same as that of <i>ja</i> , and the coefficient matrix can be either real or complex. The matrix must be stored in the 3-array variation of compressed sparse row (CSR3) format with increasing values of <i>ja</i> for each row	in
<i>ia (n+1)</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: MKL_INT*	<i>rowIndex</i> array in CSR3 format	>=0	<i>ia (i)</i> gives the index of the element in array <i>a</i> that contains the first non-zero element from row <i>i</i> of <i>A</i> . The last element <i>ia (n+1)</i> (for one-based indexing) or <i>ia (n)</i> (for zero-based indexing) is taken to be equal to the number of non-zero elements in <i>A</i> , plus one. Note: <i>iparm(35)</i> indicates whether row/column indexing starts from 1 or 0.	in

Parameter	Type	Description	Values	Comments	In/ Out
$ja(*)$	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: MKL_INT*	$columns$ array in CSR3 format	$>=0$	The column indices for each row of A must be sorted in increasing order. For structurally symmetric matrices zero diagonal elements must be stored in a and ja . Zero diagonal elements should be stored for symmetric matrices, although they are not required. For symmetric matrices, the solver needs only the upper triangular part of the system. Note: iparm(35) indicates whether row/column indexing starts from 1 or 0.	in
$perm(n)$	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(INOUT) C: MKL_INT*	Holds the permutation vector of size n or specifies elements used for computing a partial solution	$>=0$	You can apply your own fill-in reducing ordering (iparm(5) = 1) or return the permutation from the solver (iparm(5) = 2). Let $C = P^* A^* P^T$ be the permuted matrix. Row (column) i of C is the $perm(i)$ row (column) of A . The numbering of the array must describe a permutation. To specify elements for a partial solution, set $iparm(5) = 0$, $iparm(31) > 0$, and $iparm(36) = 0$. To specify elements for a Schur complement, set $iparm(5) = 0$, $iparm(31) = 0$, and $iparm(36) > 0$. Note: iparm(35) indicates whether row/column indexing starts from 1 or 0.	in/ out
$nrhs$	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: MKL_INT*	Number of right-hand sides that need to be solved for	$>=0$	Generally used value is 1 To obtain better Intel MKL PARDISO performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase.	in

Parameter	Type	Description	Values	Comments	In/ Out
<i>iparm(64)</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(INOUT) C: MKL_INT*	This array is used to pass various parameters to Intel MKL PARDISO and to return some useful information after execution of the solver (see pardiso iparm Parameter for more details)	*	If <i>iparm(1)=0</i> , Intel MKL PARDISO fills <i>iparm(2)</i> through <i>iparm(64)</i> with default values and uses them.	in/ out
<i>msglvl</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: MKL_INT*	Message level information	0 1	Intel MKL PARDISO generates no output Intel MKL PARDISO prints statistical information	in
<i>b(n*nrhs)</i>	FORTRAN 77: PARDISO_DATA_TYPE vectors 1) Fortran 90: PARDISO_DATA_TYPE 1), INTENT(INOUT) C: void*	Right-hand side	*	On entry, contains the right-hand side vector/matrix <i>B</i> , which is placed contiguously in memory. The <i>b(i+(k-1)*nrhs)</i> element must hold the <i>i</i> -th component of <i>k</i> -th right-hand side vector. Note that <i>b</i> is only accessed in the solution phase. On output, the array is replaced with the solution if <i>iparm(6)=1</i> .	in/ out
<i>x(n*nrhs)</i>	FORTRAN 77: PARDISO_DATA_TYPE 1) Fortran 90: PARDISO_DATA_TYPE 1), INTENT(OUT) C: void*	Solution vectors	*	On output, if <i>iparm(6)=0</i> , contains solution vector/matrix <i>X</i> which is placed contiguously in memory. The <i>x(i+(k-1)*nrhs)</i> element must hold the <i>i</i> -th component of <i>k</i> -th solution vector. Note that <i>x</i> is only accessed in the solution phase.	out
<i>error</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(OUT) C: MKL_INT*	Error indicator	0 -1 -2 -3 -4 -5	No error Input inconsistent Not enough memory Reordering problem Zero pivot, numerical factorization or iterative refinement problem Unclassified (internal) error	out

Parameter	Type	Description	Values	Comments	In/ Out
			-6	Reordering failed (matrix types 11 and 13 only)	
			-7	Diagonal matrix is singular	
			-8	32-bit integer overflow problem	
			-9	Not enough memory for OOC	
			-10	Problems with opening OOC temporary files	
			-11	Read/write problems with the OOC data file	

¹⁾ See description of `PARDISO_DATA_TYPE` in [PARDISO_DATA_TYPE](#).

pardiso iparm Parameter

The following table describes all individual components of the Intel MKL PARDISO `iparm()` parameter. Components which are not used must be initialized with 0. Default values are denoted with an asterisk (*).

Component	Description						
<code>iparm(1)</code>	Use default values.						
<code>input</code>	<table> <tr> <td>0</td> <td><code>iparm(2) - iparm(64)</code> are filled with default values.</td> </tr> <tr> <td>$\neq 0$</td> <td>You must supply all values in components <code>iparm(2) - iparm(64)</code>.</td> </tr> </table>	0	<code>iparm(2) - iparm(64)</code> are filled with default values.	$\neq 0$	You must supply all values in components <code>iparm(2) - iparm(64)</code> .		
0	<code>iparm(2) - iparm(64)</code> are filled with default values.						
$\neq 0$	You must supply all values in components <code>iparm(2) - iparm(64)</code> .						
<code>iparm(2)</code>	Fill-in reducing ordering for the input matrix.						
<code>input</code>	<p>CAUTION</p> <p>You can control the parallel execution of the solver by explicitly setting the <code>MKL_NUM_THREADS</code> environment variable. If fewer threads are available than specified, the execution may slow down instead of speeding up. If <code>MKL_NUM_THREADS</code> is not defined, then the solver uses all available processors.</p> <table> <tr> <td>0</td> <td>The minimum degree algorithm [Li99].</td> </tr> <tr> <td>2*</td> <td>The nested dissection algorithm from the METIS package [Karypis98].</td> </tr> <tr> <td>3</td> <td>The parallel (OpenMP) version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Intel MKL PARDISO Phase 1 takes significant time.</td> </tr> </table>	0	The minimum degree algorithm [Li99] .	2*	The nested dissection algorithm from the METIS package [Karypis98] .	3	The parallel (OpenMP) version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Intel MKL PARDISO Phase 1 takes significant time.
0	The minimum degree algorithm [Li99] .						
2*	The nested dissection algorithm from the METIS package [Karypis98] .						
3	The parallel (OpenMP) version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Intel MKL PARDISO Phase 1 takes significant time.						
<code>iparm(3)</code>	Reserved. Set to zero.						
<code>iparm(4)</code>	Preconditioned CGS/CG.						
<code>input</code>	This parameter controls preconditioned CGS [Sonn89] for nonsymmetric or structurally symmetric matrices and Conjugate-Gradients for symmetric matrices. <code>iparm(4)</code> has the form $iparm(4) = 10 * L + K$.						
$K=0$	The factorization is always computed as required by <code>phase</code> .						
$K=1$	CGS iteration replaces the computation of LU . The preconditioner is LU that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.						

Component	Description								
K=2	CGS iteration for symmetric matrices replaces the computation of LU . The preconditioner is LU that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.								
	The value L controls the stopping criterion of the Krylow-Subspace iteration: $\text{eps}_{\text{CGS}} = 10^{-L}$ is used in the stopping criterion $\ dx_i\ / \ dx_0\ < \text{eps}_{\text{CGS}}$ where $\ dx_i\ = \ \text{inv}(L^*U)^*r_i\ $ and r_i is the residue at iteration i of the preconditioned Krylow-Subspace iteration. A maximum number of 150 iterations is fixed with the assumption that the iteration will converge before consuming half the factorization time. Intermediate convergence rates and residue excursions are checked and can terminate the iteration process. If $\text{phase} = 23$, then the factorization for a given A is automatically recomputed in cases where the Krylow-Subspace iteration failed, and the corresponding direct solution is returned. Otherwise the solution from the preconditioned Krylow-Subspace iteration is returned. Using $\text{phase} = 33$ results in an error message ($\text{error} = -4$) if the stopping criteria for the Krylow-Subspace iteration can not be reached. More information on the failure can be obtained from iparm(20) . The default is $\text{iparm}(4)=0$, and other values are only recommended for an advanced user. $\text{iparm}(4)$ must be greater or equal to zero.								
	Examples:								
	<table> <thead> <tr> <th><code>iparm(4)</code></th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>31</td> <td>LU-preconditioned CGS iteration with a stopping criterion of 1.0E-3 for nonsymmetric matrices</td> </tr> <tr> <td>61</td> <td>LU-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for nonsymmetric matrices</td> </tr> <tr> <td>62</td> <td>LU-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric matrices</td> </tr> </tbody> </table>	<code>iparm(4)</code>	Description	31	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-3 for nonsymmetric matrices	61	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for nonsymmetric matrices	62	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric matrices
<code>iparm(4)</code>	Description								
31	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-3 for nonsymmetric matrices								
61	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for nonsymmetric matrices								
62	LU -preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric matrices								
<code>iparm(5)</code>	User permutation.								
input	This parameter controls whether user supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithms. Another use of this parameter is to control obtaining the fill-in reducing permutation vector calculated during the reordering stage of Intel MKL PARDISO. This option is useful for testing reordering algorithms, adapting the code to special applications problems (for instance, to move zero diagonal elements to the end of $P^*A^*P^T$), or for using the permutation vector more than once for matrices with identical sparsity structures. For definition of the permutation, see the description of the <code>perm</code> parameter.								
CAUTION									
You can only set one of <code>iparm(5)</code> , <code>iparm(31)</code> , and <code>iparm(36)</code> , so be sure that the <code>iparm(31)</code> (partial solution) and the <code>iparm(36)</code> (Schur complement) parameters are 0 if you set <code>iparm(5)</code> .									
0*	User permutation in the <code>perm</code> array is ignored.								
1	Intel MKL PARDISO uses the user supplied fill-in reducing permutation from the <code>perm</code> array. <code>iparm(2)</code> is ignored.								
NOTE									
Setting <code>iparm(5) = 1</code> prevents use of a parallel algorithm for the solve step.									

Component	Description
2	Intel MKL PARDISO returns the permutation vector computed at phase 1 in the <i>perm</i> array.
<i>iparm(6)</i> input	Write solution on <i>x</i> . NOTE The array <i>x</i> is always used.
0*	The array <i>x</i> contains the solution; right-hand side vector <i>b</i> is kept unchanged.
1	The solver stores the solution on the right-hand side <i>b</i> .
<i>iparm(7)</i> output	Number of iterative refinement steps performed. Reports the number of iterative refinement steps that were actually performed during the solve step.
<i>iparm(8)</i> input	Iterative refinement step. On entry to the solve and iterative refinement step, <i>iparm(8)</i> must be set to the maximum number of iterative refinement steps that the solver performs. 0* The solver automatically performs two steps of iterative refinement when perturbed pivots are obtained during the numerical factorization. >0 Maximum number of iterative refinement steps that the solver performs. The solver performs not more than the absolute value of <i>iparm(8)</i> steps of iterative refinement. The solver might stop the process before the maximum number of steps if <ul style="list-style-type: none"> • a satisfactory level of accuracy of the solution in terms of backward error is achieved, • or if it determines that the required accuracy cannot be reached. In this case Intel MKL PARDISO returns -4 in the <i>error</i> parameter. The number of executed iterations is reported in <i>iparm(7)</i> . <0 Same as above, but the accumulation of the residuum uses extended precision real and complex data types. Perturbed pivots result in iterative refinement (independent of <i>iparm(8)=0</i>) and the number of executed iterations is reported in <i>iparm(7)</i> .
<i>iparm(9)</i>	Reserved. Set to zero.
<i>iparm(10)</i> input	Pivoting perturbation. This parameter instructs Intel MKL PARDISO how to handle small pivots or zero pivots for nonsymmetric matrices (<i>mtype</i> =11 or <i>mtype</i> =13) and symmetric matrices (<i>mtype</i> =-2, <i>mtype</i> =-4, or <i>mtype</i> =6). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04]. Small pivots are perturbed with $\text{eps} = 10^{-\text{iparm}(10)}$. The magnitude of the potential pivot is tested against a constant threshold of $\text{alpha} = \text{eps} * \ A2\ _{\text{inf}}$,

Component	Description
	where $\text{eps} = 10^{(-\text{iparm}(10))}$, $A2 = P^* P_{\text{MPS}}^* D_r^* A^* D_c^* P$, and $\ A2\ _{\text{inf}}$ is the infinity norm of the scaled and permuted matrix A. Any tiny pivots encountered during elimination are set to the sign $(\text{LII})^* \text{eps} * \ A2\ _{\text{inf}}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with $\text{eps} = 10^{(-\text{iparm}(10))}$.
13*	The default value for nonsymmetric matrices ($mtype = 11$, $mtype=13$), $\text{eps} = 10^{-13}$.
8*	The default value for symmetric indefinite matrices ($mtype = -2$, $mtype=-4$, $mtype=6$), $\text{eps} = 10^{-8}$.
<i>iparm(11)</i>	Scaling vectors.
input	<p>Intel MKL PARDISO uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less than or equal to 1. This scaling method is applied only to nonsymmetric matrices ($mtype = 11$ or $mtype = 13$). The scaling can also be used for symmetric indefinite matrices ($mtype = -2$, $mtype = -4$, or $mtype = 6$) when the symmetric weighted matchings are applied (<i>iparm(13)</i> = 1).</p> <p>Use <i>iparm(11)</i> = 1 (scaling) and <i>iparm(13)</i> = 1 (matching) for highly indefinite symmetric matrices, for example, from interior point optimizations or saddle point problems. Note that in the analysis phase (<i>phase</i>=11) you must provide the numerical values of the matrix A in array <i>a</i> in case of scaling and symmetric weighted matching.</p>
0*	Disable scaling. Default for symmetric indefinite matrices.
1*	<p>Enable scaling. Default for nonsymmetric matrices.</p> <p>Scale the matrix so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less or equal to 1. This scaling method is applied to nonsymmetric matrices ($mtype = 11$, $mtype = 13$). The scaling can also be used for symmetric indefinite matrices ($mtype = -2$, $mtype = -4$, $mtype = 6$) when the symmetric weighted matchings are applied (<i>iparm(13)</i> = 1).</p> <p>Note that in the analysis phase (<i>phase</i>=11) you must provide the numerical values of the matrix A in case of scaling.</p>
<i>iparm(12)</i>	Solve with transposed or conjugate transposed matrix A.
input	<p>NOTE For real matrices the terms <i>transposed</i> and <i>conjugate transposed</i> are equivalent.</p>
0*	Solve a linear system $AX = B$.
1	Solve a conjugate transposed system $A^H X = B$ based on the factorization of the matrix A.
2	Solve a transposed system $A^T X = B$ based on the factorization of the matrix A.
<i>iparm(13)</i>	Improved accuracy using (non-) symmetric weighted matching.
input	Intel MKL PARDISO can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to the factorization methods and complements the alternative of using more complete pivoting techniques during the numerical factorization.
0*	Disable matching. Default for symmetric indefinite matrices.

Component	Description				
1*	<p>Enable matching. Default for nonsymmetric matrices.</p> <p>Maximum weighted matching algorithm to permute large elements close to the diagonal.</p> <p>It is recommended to use <code>iparm(11) = 1</code> (scaling) and <code>iparm(13) = 1</code> (matching) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems.</p> <p>Note that in the analysis phase (<code>phase=11</code>) you must provide the numerical values of the matrix A in case of symmetric weighted matching.</p>				
<code>iparm(14)</code>	Number of perturbed pivots.				
<code>output</code>	After factorization, contains the number of perturbed pivots for the matrix types: 11, 13, -2, -4 and -6.				
<code>iparm(15)</code>	Peak memory on symbolic factorization.				
<code>output</code>	The total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase. This value is only computed in phase 1.				
<code>iparm(16)</code>	Permanent memory on symbolic factorization.				
<code>output</code>	Permanent memory from the analysis and symbolic factorization phase in kilobytes that the solver needs in the factorization and solve phases. This value is only computed in phase 1.				
<code>iparm(17)</code>	Size of factors/Peak memory on numerical factorization and solution.				
<code>output</code>	This parameter provides the size in kilobytes of the total memory consumed by in-core Intel MKL PARDISO for internal floating point arrays. This parameter is computed in phase 1. See <code>iparm(63)</code> for the OOC mode. The total peak memory consumed by Intel MKL PARDISO is $\max(iparm(15), iparm(16)+iparm(17))$.				
<code>iparm(18)</code>	Report the number of non-zero elements in the factors.				
<code>input/output</code>	<table border="0"> <tr> <td><0</td> <td>Enable reporting if <code>iparm(18) < 0</code> on entry. The default value is -1.</td> </tr> <tr> <td>≥ 0</td> <td>Disable reporting.</td> </tr> </table>	<0	Enable reporting if <code>iparm(18) < 0</code> on entry. The default value is -1.	≥ 0	Disable reporting.
<0	Enable reporting if <code>iparm(18) < 0</code> on entry. The default value is -1.				
≥ 0	Disable reporting.				
<code>iparm(19)</code>	Report number of floating point operations (in 10^6 floating point operations) that are necessary to factor the matrix A .				
<code>input/output</code>	<table border="0"> <tr> <td><0</td> <td>Enable report if <code>iparm(19) < 0</code> on entry. This increases the reordering time.</td> </tr> <tr> <td>≥ 0</td> <td>Disable report.</td> </tr> </table>	<0	Enable report if <code>iparm(19) < 0</code> on entry. This increases the reordering time.	≥ 0	Disable report.
<0	Enable report if <code>iparm(19) < 0</code> on entry. This increases the reordering time.				
≥ 0	Disable report.				
<code>iparm(20)</code>	Report CG/CGS diagnostics.				
<code>output</code>	<table border="0"> <tr> <td>> 0</td> <td>CGS succeeded, reports the number of completed iterations.</td> </tr> <tr> <td>< 0</td> <td>CG/CGS failed (<code>error=-4</code> after the solution phase). If <code>phase= 23</code>, then the factors L and U are recomputed for the matrix A and the error flag <code>error=0</code> in case of a successful factorization. If <code>phase = 33</code>, then <code>error = -4</code> signals failure. <code>iparm(20)= - it_cgs*10 - cgs_error.</code> Possible values of <code>cgs_error</code>: 1 - fluctuations of the residuum are too large</td> </tr> </table>	> 0	CGS succeeded, reports the number of completed iterations.	< 0	CG/CGS failed (<code>error=-4</code> after the solution phase). If <code>phase= 23</code> , then the factors L and U are recomputed for the matrix A and the error flag <code>error=0</code> in case of a successful factorization. If <code>phase = 33</code> , then <code>error = -4</code> signals failure. <code>iparm(20)= - it_cgs*10 - cgs_error.</code> Possible values of <code>cgs_error</code> : 1 - fluctuations of the residuum are too large
> 0	CGS succeeded, reports the number of completed iterations.				
< 0	CG/CGS failed (<code>error=-4</code> after the solution phase). If <code>phase= 23</code> , then the factors L and U are recomputed for the matrix A and the error flag <code>error=0</code> in case of a successful factorization. If <code>phase = 33</code> , then <code>error = -4</code> signals failure. <code>iparm(20)= - it_cgs*10 - cgs_error.</code> Possible values of <code>cgs_error</code> : 1 - fluctuations of the residuum are too large				

Component	Description
	<p>2 - $\ \mathbf{dx}_{\text{max_it_cgs}/2} \$ is too large (slow convergence)</p> <p>3 - stopping criterion is not reached at max_it_cgs</p> <p>4 - perturbed pivots caused iterative refinement</p> <p>5 - factorization is too fast for this matrix. It is better to use the factorization method with <code>iparm(4) = 0</code></p>
<code>iparm(21)</code> input	Pivoting for symmetric indefinite matrices.
	<p>NOTE</p> <p>Use <code>iparm(11) = 1</code> (scaling) and <code>iparm(13) = 1</code> (matchings) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems.</p>
0	Apply 1x1 diagonal pivoting during the factorization process.
1*	Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of $mtype=-2$, $mtype=-4$, or $mtype=6$.
2	Apply 1x1 diagonal pivoting during the factorization process. Using this value is the same as using <code>iparm(21) = 0</code> except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by <code>iparm(8)</code> (0 by default).
3	Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of $mtype=-2$, $mtype=-4$, or $mtype=6$. Using this value is the same as using <code>iparm(21) = 1</code> except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by <code>iparm(8)</code> (0 by default).
<code>iparm(22)</code> output	Inertia: number of positive eigenvalues. Intel MKL PARDISO reports the number of positive eigenvalues for symmetric indefinite matrices.
<code>iparm(23)</code> output	Inertia: number of negative eigenvalues. Intel MKL PARDISO reports the number of negative eigenvalues for symmetric indefinite matrices.
<code>iparm(24)</code> input	Parallel factorization control.
	<p>NOTE</p> <p>The two-level factorization algorithm does not improve performance in OOC mode.</p>
0*	Intel MKL PARDISO uses the classic algorithm for factorization.
1	Intel MKL PARDISO uses a two-level factorization algorithm. This algorithm generally improves scalability in case of parallel factorization on many threads (more than eight).
<code>iparm(25)</code> input	Parallel forward/backward solve control.
	0* Intel MKL PARDISO uses a parallel algorithm for the solve step.

Component	Description
1	Intel MKL PARDISO uses the sequential forward and backward solve. This feature is available only for in-core Intel MKL PARDISO (see iparm(60)).
<i>iparm</i> (26)	Reserved. Set to zero.
<i>iparm</i> (27)	Matrix checker.
input	<p>0* Intel MKL PARDISO does not check the sparse matrix representation for errors.</p> <p>1 Intel MKL PARDISO checks integer arrays <i>ia</i> and <i>ja</i>. In particular, Intel MKL PARDISO checks whether column indices are sorted in increasing order within each row.</p>
<i>iparm</i> (28)	Single or double precision Intel MKL PARDISO.
input	See iparm(8) for information on controlling the precision of the refinement steps.
Important	
The <i>iparm</i> (28) value is stored in the Intel MKL PARDISO handle between Intel MKL PARDISO calls, so the precision mode can be changed only during phase 1.	
0*	Input arrays (<i>a</i> , <i>x</i> and <i>b</i>) and all internal arrays must be presented in double precision.
1	Input arrays (<i>a</i> , <i>x</i> and <i>b</i>) must be presented in single precision. In this case all internal computations are performed in single precision.
<i>iparm</i> (29)	Reserved. Set to zero.
<i>iparm</i> (30)	Number of zero or negative pivots.
output	If Intel MKL PARDISO detects zero or negative pivot for <i>mtype</i> =2 or <i>mtype</i> =4 matrix types, the factorization is stopped, Intel MKL PARDISO returns immediately with an <i>error</i> = -4, and <i>iparm</i> (30) reports the number of the equation where the first zero or negative pivot is detected.
<i>iparm</i> (31)	Partial solve and computing selected components of the solution vectors.
input	This parameter controls the solve step of Intel MKL PARDISO. It can be used if only a few components of the solution vectors are needed or if you want to reduce the computation cost at the solve step by utilizing the sparsity of the right-hand sides. To use this option the input permutation vector define <i>perm</i> so that when <i>perm</i> (<i>i</i>) = 1 it means that either the <i>i</i> -th component in the right-hand sides is nonzero, or the <i>i</i> -th component in the solution vectors is computed, or both, depending on the value of <i>iparm</i> (31). The permutation vector <i>perm</i> must be present in all phases of Intel MKL PARDISO software. At the reordering step, the software overwrites the input vector <i>perm</i> by a permutation vector used by the software at the factorization and solver step. If <i>m</i> is the number of components such that <i>perm</i> (<i>i</i>) = 1, then the last <i>m</i> components of the output vector <i>perm</i> are a set of the indices <i>i</i> satisfying the condition <i>perm</i> (<i>i</i>) = 1 on input.
NOTE	
Turning on this option often increases the time used by Intel MKL PARDISO for factorization and reordering steps, but it can reduce the time required for the solver step.	

Component	Description
	<p>Important This feature is only available for in-core Intel MKL PARDISO, so to use it you must set <code>iparm(60) = 0</code>. Set the parameters <code>iparm(8)</code> (iterative refinement steps), <code>iparm(4)</code> (preconditioned CGS), <code>iparm(5)</code> (user permutation), and <code>iparm(36)</code> (Schur complement) to 0 as well.</p>
0*	Disables this option.
1	<p>it is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector <code>perm</code> is defined so that <code>perm(i) = 1</code> means that the (i)-th component in the right-hand sides is nonzero. In this case Intel MKL PARDISO only uses the non-zero components of the right-hand side vectors and computes only corresponding components in the solution vectors. That means the i-th component in the solution vectors is only computed if <code>perm(i) = 1</code>.</p>
2	<p>It is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector <code>perm</code> is defined so that <code>perm(i) = 1</code> means that the i-th component in the right-hand sides is nonzero.</p> <p>Unlike for <code>iparm(31)=1</code>, all components of the solution vector are computed for this setting and all components of the right-hand sides are used. Because all components are used, for <code>iparm(31)=2</code> you must set the i-th component of the right-hand sides to zero explicitly if <code>perm(i)</code> is not equal to 1.</p>
3	Selected components of the solution vectors are computed. The <code>perm</code> array is not related to the right-hand sides and it only indicates which components of the solution vectors should be computed. In this case <code>perm(i) = 1</code> means that the i -th component in the solution vectors is computed.
<code>iparm(32)</code>	Reserved. Set to zero.
<code>iparm(33)</code>	
<code>iparm(34)</code>	Optimal number of threads for conditional numerical reproducibility (CNR) mode.
input	<p>Intel MKL PARDISO reads the value of <code>iparm(34)</code> during the analysis phase (phase 1), so you cannot change it later.</p> <p>Because Intel MKL PARDISO uses C random number generator facilities during the analysis phase (phase 1) you must take these precautions to get numerically reproducible results:</p> <ul style="list-style-type: none"> • Do not alter the states of the random number generators. • Do not run multiple instances of Intel MKL PARDISO in parallel in the analysis phase (phase 1).
NOTE	
<p>CNR is only available for the in-core version of Intel MKL PARDISO, so you must also set <code>iparm(60)</code> to 0 in order to use the in-core version. Otherwise Intel MKL PARDISO does not produce numerically repeatable results even if CNR is enabled for Intel MKL using the functionality described in Support Functions for CNR.</p>	
0*	CNR mode for Intel MKL PARDISO is enabled only if it is enabled for Intel MKL using the functionality described in Support Functions for CNR and the in-core version is used. Intel MKL PARDISO determines the optimal number of threads automatically, and produces numerically reproducible results regardless of the number of threads.

Component	Description				
	>0 CNR mode is enabled for Intel MKL PARDISO if in-core version is used and the optimal number of threads for Intel MKL PARDISO to rely on is defined by the value of <i>iparm(34)</i> . You can use <i>iparm(34)</i> to enable CNR mode independent from other Intel MKL domains. To get the best performance, set <i>iparm(34)</i> to the actual number of hardware threads dedicated for Intel MKL PARDISO. Setting <i>iparm(34)</i> to fewer threads than the maximum number of threads in use reduces the scalability of the problem being solved. Setting <i>iparm(34)</i> to more threads than are available can reduce the performance of Intel MKL PARDISO.				
<i>iparm(35)</i>	One- or zero-based indexing of columns and rows.				
input	<table border="0"> <tr> <td>0*</td> <td>One-based indexing: columns and rows indexing in arrays <i>ia</i>, <i>ja</i>, and <i>perm</i> starts from 1 (Fortran-style indexing).</td> </tr> <tr> <td>1</td> <td>Zero-based indexing: columns and rows indexing in arrays <i>ia</i>, <i>ja</i>, and <i>perm</i> starts from 0 (C-style indexing).</td> </tr> </table>	0*	One-based indexing: columns and rows indexing in arrays <i>ia</i> , <i>ja</i> , and <i>perm</i> starts from 1 (Fortran-style indexing).	1	Zero-based indexing: columns and rows indexing in arrays <i>ia</i> , <i>ja</i> , and <i>perm</i> starts from 0 (C-style indexing).
0*	One-based indexing: columns and rows indexing in arrays <i>ia</i> , <i>ja</i> , and <i>perm</i> starts from 1 (Fortran-style indexing).				
1	Zero-based indexing: columns and rows indexing in arrays <i>ia</i> , <i>ja</i> , and <i>perm</i> starts from 0 (C-style indexing).				
<i>iparm(36)</i>	Schur complement matrix computation control. To calculate this matrix, you must set the input permutation vector <i>perm</i> to a set of indexes such that when <i>perm(i) = 1</i> , the <i>i</i> -th element of the initial matrix is an element of the Schur matrix.				
	CAUTION You can only set one of <i>iparm(5)</i> , <i>iparm(31)</i> , and <i>iparm(36)</i> , so be sure that the <i>iparm(5)</i> (user permutation) and the <i>iparm(31)</i> (partial solution) parameters are 0 if you set <i>iparm(36)</i> .				
	<table border="0"> <tr> <td>0*</td> <td>Do not compute Schur complement.</td> </tr> <tr> <td>1</td> <td>Compute Schur complement matrix as part of Intel MKL PARDISO factorization step and return it in the solution vector.</td> </tr> </table>	0*	Do not compute Schur complement.	1	Compute Schur complement matrix as part of Intel MKL PARDISO factorization step and return it in the solution vector.
0*	Do not compute Schur complement.				
1	Compute Schur complement matrix as part of Intel MKL PARDISO factorization step and return it in the solution vector.				
	NOTE This option only computes the Schur complement matrix, and does not calculate factorization arrays.				
	<table border="0"> <tr> <td>2</td> <td>Compute Schur complement matrix as part of Intel MKL PARDISO factorization step and return it in the solution vector. Since this option calculates factorization arrays you can use it to launch partial or full solution of the entire problem after the factorization step.</td> </tr> </table>	2	Compute Schur complement matrix as part of Intel MKL PARDISO factorization step and return it in the solution vector. Since this option calculates factorization arrays you can use it to launch partial or full solution of the entire problem after the factorization step.		
2	Compute Schur complement matrix as part of Intel MKL PARDISO factorization step and return it in the solution vector. Since this option calculates factorization arrays you can use it to launch partial or full solution of the entire problem after the factorization step.				
<i>iparm(37)</i>	Reserved. Set to zero.				
<i>iparm(55)</i>					
<i>iparm(56)</i>	Diagonal and pivoting control.				
	<table border="0"> <tr> <td>0*</td> <td>Internal function used to work with pivot and calculation of diagonal arrays turned off.</td> </tr> <tr> <td>1</td> <td>You can use the <i>mkl_pardiso_pivot</i> callback routine to control pivot elements which appear during numerical factorization. Additionally, you can obtain the elements of initial matrix and factorized matrices after the <i>pardiso</i> factorization step diagonal using the <i>pardiso_getdiag</i> routine. This parameter can be turned on only in the in-core version of Intel MKL PARDISO.</td> </tr> </table>	0*	Internal function used to work with pivot and calculation of diagonal arrays turned off.	1	You can use the <i>mkl_pardiso_pivot</i> callback routine to control pivot elements which appear during numerical factorization. Additionally, you can obtain the elements of initial matrix and factorized matrices after the <i>pardiso</i> factorization step diagonal using the <i>pardiso_getdiag</i> routine. This parameter can be turned on only in the in-core version of Intel MKL PARDISO.
0*	Internal function used to work with pivot and calculation of diagonal arrays turned off.				
1	You can use the <i>mkl_pardiso_pivot</i> callback routine to control pivot elements which appear during numerical factorization. Additionally, you can obtain the elements of initial matrix and factorized matrices after the <i>pardiso</i> factorization step diagonal using the <i>pardiso_getdiag</i> routine. This parameter can be turned on only in the in-core version of Intel MKL PARDISO.				
<i>iparm(57)</i>	Reserved. Set to zero.				
<i>iparm(59)</i>					

Component	Description
<i>iparm</i> (60)	Intel MKL PARDISO mode.
input	<p><i>iparm</i>(60) switches between in-core (IC) and out-of-core (OOC) Intel MKL PARDISO. OOC can solve very large problems by holding the matrix factors in files on the disk, which requires a reduced amount of main memory compared to IC.</p> <p>Unless you are operating in sequential mode, you can switch between IC and OOC modes after the reordering phase. However, you can get better Intel MKL PARDISO performance by setting <i>iparm</i>(60) before the reordering phase.</p>
NOTE	
The amount of memory used in OOC mode depends on the number of threads.	
WARNING	
Do not increase the number of threads used for Intel MKL PARDISO between the first call to <code>pardiso</code> and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of threads, increasing it after the initial call can cause incorrect results.	
0*	IC mode.
1	<p>IC mode is used if the total amount of RAM (in megabytes) needed for storing the matrix factors is less than sum of two values of the environment variables: <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code> (default value 2000 MB) and <code>MKL_PARDISO_OOC_MAX_SWAP_SIZE</code> (default value 0 MB); otherwise OOC mode is used. In this case amount of RAM used by OOC mode cannot exceed the value of <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code>.</p> <p>If the total peak memory needed for storing the local arrays is more than <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code>, increase <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code> if possible.</p>
2	<p>OOC mode.</p> <p>The OOC mode can solve very large problems by holding the matrix factors in files on the disk. Hence the amount of RAM required by OOC mode is significantly reduced compared to IC mode.</p> <p>If the total peak memory needed for storing the local arrays is more than <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code>, increase <code>MKL_PARDISO_OOC_MAX_CORE_SIZE</code> if possible.</p> <p>To obtain better Intel MKL PARDISO performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase.</p>
<i>iparm</i> (61)	Reserved. Set to zero.
-	
<i>iparm</i> (62)	
<i>iparm</i> (63)	Size of the minimum OOC memory for numerical factorization and solution.
output	<p>This parameter provides the size in kilobytes of the minimum memory required by OOC Intel MKL PARDISO for internal floating point arrays. This parameter is computed in phase 1.</p> <p>Total peak memory consumption of OOC Intel MKL PARDISO can be estimated as $\max(\text{iparm}(15), \text{iparm}(16) + \text{iparm}(63))$.</p>
<i>iparm</i> (64)	Reserved. Set to zero.

NOTE

Generally in sparse matrices, components which are equal to zero can be considered non-zero if necessary. For example, in order to make a matrix structurally symmetric, elements which are zero can be considered non-zero. See [Sparse Matrix Storage Formats](#) for an example.

PARDISO_DATA_TYPE

The following table lists the values of `PARDISO_DATA_TYPE` depending on the matrix types and values of the parameter `iparm(28)`.

Data type value	Matrix type <i>mtype</i>	<i>iparm(28)</i>	comments
DOUBLE PRECISION	1, 2, -2, 11	0	Real matrices, double precision
REAL		1	Real matrices, single precision
DOUBLE COMPLEX	3, 6, 13, 4, -4	0	Complex matrices, double precision
COMPLEX		1	Complex matrices, single precision

Parallel Direct Sparse Solver for Clusters Interface

The Parallel Direct Sparse Solver for Clusters Interface solves large linear systems of equations with sparse matrices on clusters. It is

- high performing
- robust
- memory efficient
- easy to use

A hybrid implementation combines Message Passing Interface (MPI) technology for data exchange between parallel tasks (processes) running on different nodes, and OpenMP* technology for parallelism inside each node of the cluster. This approach effectively uses modern hardware resources such as clusters consisting of nodes with multi-core processors. The solver code is optimized for the latest Intel processors, but also performs well on clusters consisting of non-Intel processors.

Parallel Direct Sparse Solver for Clusters Interface provides a Fortran interface, but can be called from C programs by observing Fortran parameter passing and naming conventions used by the supported compilers and operating systems. Code examples are available in the Intel MKL installation examples directory.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Supported Configuration

Parallel Direct Sparse Solver for Clusters Interface supports Intel® 64 architecture and compatible architectures with 64-bit Linux* and an ILP64/LP64 interface.

Parallel Direct Sparse Solver for Clusters Interface supports these MPI implementations:

- Intel® MPI Library 3.0 or later
- MPICH2 1.2.1
- Open MPI 1.2.6
- MVAPICH2 1.2

Parallel Direct Sparse Solver for Clusters Interface Algorithm

Parallel Direct Sparse Solver for Clusters Interface solves a set of sparse linear equations

$$A^*X = B$$

with multiple right-hand sides using a distributed LU , LL^T , LDL^T or LDL^* factorization, where A is an n -by- n matrix, and X and B are n -by- $nrhs$ matrices.

The solution comprises four tasks:

- analysis and symbolic factorization;
- numerical factorization;
- forward and backward substitution including iterative refinement;
- termination to release all internal solver memory.

The solver first computes a symmetric fill-in reducing permutation P based on the nested dissection algorithm from the METIS package [Karypis98] (included with Intel MKL), followed by the Cholesky or other type of factorization (depending on matrix type) [Schenk00-2] of PAP^T . The solver uses either diagonal pivoting, or 1x1 and 2x2 Bunch and Kaufman pivoting for symmetric indefinite or Hermitian matrices before finding an approximation of X by forward and backward substitution and iterative refinement.

The initial matrix A is perturbed whenever numerically acceptable 1x1 and 2x2 pivots cannot be found within the diagonal blocks. One or two passes of iterative refinement may be required to correct the effect of the perturbations. This restricted notion of pivoting with iterative refinement is effective for highly indefinite symmetric systems. For a large set of matrices from different application areas, the accuracy of this method is comparable to a direct factorization method that uses complete sparse pivoting techniques [Schenk04].

Parallel Direct Sparse Solver for Clusters additionally improves the pivoting accuracy by applying symmetric weighted matching algorithms. These methods identify large entries in the coefficient matrix A that, if permuted close to the diagonal, enable the factorization process to identify more acceptable pivots and proceed with fewer pivot perturbations. The methods are based on maximum weighted matching and improve the quality of the factor in a complementary way to the alternative idea of using more complete pivoting techniques.

Parallel Direct Sparse Solver for Clusters Interface Matrix Storage

The sparse data storage in the Parallel Direct Sparse Solver for Clusters Interface follows the scheme described in the [Sparse Matrix Storage Formats](#) section using the variable *ja* for *columns*, *ia* for *rowIndex*, and *a* for *values*. Column indices *ja* must be in increasing order per row and each diagonal element to be present in the matrix structure (even if it is zero).

When an input data structure is not accessed in a call, a NULL pointer or any valid address can be passed as a placeholder for that argument.

Algorithm Parallelization and Data Distribution

Intel® MKL Parallel Direct Sparse Solver for Clusters enables parallel execution of the solution algorithm with efficient data distribution.

The master MPI process performs the symbolic factorization phase to represent matrix A as computational tree. Then matrix A is divided among all MPI processes in a one-dimensional manner. The same distribution is used for *L-factor* (the lower triangular matrix in Cholesky decomposition). Matrix A and all required internal data are broadcast to slave MPI processes. Each MPI process fills in its own parts of *L-factor* with initial values of the matrix A .

Parallel Direct Sparse Solver for Clusters Interface computes all independent parts of *L-factor* completely in parallel. When a block of the factor must be updated by other blocks, these updates are independently passed to a temporary array on each updating MPI process. It further gathers the result into an updated block using the `MPI_Reduce()` routine. The computations within an MPI process are dynamically divided among threads using pipelining parallelism with a combination of left- and right-looking techniques similar to those of the PARDISO* software. Level 3 BLAS operations from Intel MKL ensure highly efficient performance of block-to-block update operations.

During forward/backward substitutions, respective Right Hand Side (RHS) parts are distributed among all MPI processes. All these processes participate in the computation of the solution. Finally, the solution is gathered on the master MPI process.

This approach demonstrates good scalability on clusters with Infiniband* technology. Another advantage of the approach is the effective distribution of *L-factor* among cluster nodes. This enables the solution of tasks with a much higher number of non-zero elements than it is possible with any Symmetric Multiprocessing (SMP) in-core direct solver.

The algorithm ensures that the memory required to keep internal data on each MPI process is decreased when the number of MPI processes in a run increases. However, the solver requires that matrix A and some other internal arrays completely fit into the memory of each MPI process.

To get the best performance, run one MPI process per physical node and set the number of OpenMP* threads per node equal to the number of physical cores on the node.

NOTE

Instead of calling `MPI_Init()`, initialize MPI with `MPI_Init_thread()` and set the MPI threading level to `MPI_THREAD_FUNNELED` or higher. For details, see the code examples in `<install_dir>/examples`.

cluster_sparse_solver

Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax

Fortran:

```
call cluster_sparse_solver (pt, maxfct, mnum, mtype, phase, n, a, ia, ja, perm, nrhs,
iparm, msglvl, b, x, comm, error)
```

C:

```
void cluster_sparse_solver (_MKL_DSS_HANDLE_t pt, MKL_INT *maxfct, MKL_INT *mnum,
MKL_INT *mtype, MKL_INT *phase, MKL_INT *n, void *a, MKL_INT *ia, MKL_INT *ja, MKL_INT
*perm, MKL_INT *nrhs, MKL_INT *iparm, MKL_INT *msglvl, void *b, void *x, MKL_INT
*comm, MKL_INT *error);
```

Include Files

- Fortran: `mkl_cluster_sparse_solver.f77`
- Fortran 90: `mkl_cluster_sparse_solver.f90`

- C: mkl_cluster_sparse_solver.h

Description

The routine `cluster_sparse_solver` calculates the solution of a set of sparse linear equations

$$A^*X = B$$

with single or multiple right-hand sides, using a parallel LU , LDL , or LL^T factorization, where A is an n -by- n matrix, and X and B are n -by- $nrhs$ vectors or matrices.

Input Parameters

NOTE

Most of the input parameters (except for the `pt`, `phase`, and `comm` parameters and, for the distributed format, the `a`, `ia`, and `ja` arrays) must be set on the master MPI process only, and ignored on other processes. Other MPI processes get all required data from the master MPI process using the MPI communicator, `comm`.

NOTE

The types given for parameters in this section are specified in FORTRAN 77 notation.

`pt` INTEGER*8 for 64-bit architectures
 Array of size 64.
 Handle to internal data structure. The entries must be set to zero before the first call to `cluster_sparse_solver`.

CAUTION

After the first call to `cluster_sparse_solver` do not modify `pt`, as that could cause a serious memory leak.

`maxfct` INTEGER
 Ignored; assumed equal to 1.

`mnum` INTEGER
 Ignored; assumed equal to 1.

`mtype` INTEGER
 Defines the matrix type, which influences the pivoting method. The Parallel Direct Sparse Solver for Clusters solver supports the following matrices:

1	real and structurally symmetric
2	real and symmetric positive definite
-2	real and symmetric indefinite
3	complex and structurally symmetric
4	complex and Hermitian positive definite

-4	complex and Hermitian indefinite
6	complex and symmetric
11	real and nonsymmetric
13	complex and nonsymmetric

phase INTEGER

Controls the execution of the solver. Usually it is a two- or three-digit integer. The first digit indicates the starting phase of execution and the second digit indicates the ending phase. Parallel Direct Sparse Solver for Clusters has the following phases of execution:

- Phase 1: Fill-reduction analysis and symbolic factorization
- Phase 2: Numerical factorization
- Phase 3: Forward and Backward solve including optional iterative refinement
- Memory release (*phase*= -1)

If a previous call to the routine has computed information from previous phases, execution may start at any phase. The *phase* parameter can have the following values:

<i>phase</i>	Solver Execution Steps
11	Analysis
12	Analysis, numerical factorization
13	Analysis, numerical factorization, solve, iterative refinement
22	Numerical factorization
23	Numerical factorization, solve, iterative refinement
33	Solve, iterative refinement
-1	Release all internal memory for all matrices

n INTEGER

Number of equations in the sparse linear systems of equations $A \cdot X = B$.
Constraint: $n > 0$.

a DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Parallel Direct Sparse Solver for Clusters Interface (*iparm*(28)=0)
REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Parallel Direct Sparse Solver for Clusters Interface (*iparm*(28)=1)
DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Parallel Direct Sparse Solver for Clusters Interface (*iparm*(28)=0)

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Parallel Direct Sparse Solver for Clusters Interface (*iparm*(28)=1)

Array. Contains the non-zero elements of the coefficient matrix *A* corresponding to the indices in *ja*. The size of *a* is the same as that of *ja* and the coefficient matrix can be either real or complex. The matrix must be stored in the three-array variant of the compressed sparse row (CSR3) format with increasing values of *ja* for each row. Refer to *values* array description in [Storage Formats for the Direct Sparse Solvers](#) for more details.

NOTE

For centralized input (*iparm*(40)=0), provide the *a* array for the master MPI process only. For distributed assembled input (*iparm*(40)=1 or *iparm*(40)=2), provide it for all MPI processes.

Important

The column indices of non-zero elements of each row of the matrix *A* must be stored in increasing order, and the diagonal elements must be available and stored in the array, even if they are zero.

ia

INTEGER

Array, dimension $(n+1)$. For $i \leq n$, *ia*(*i*) points to the first column index of row *i* in the array *ja* in compressed sparse row format. That is, *ia*(*i*) gives the index of the element in array *a* that contains the first non-zero element from row *i* of *A*. The last element *ia*($n+1$) is taken to be equal to the number of non-zero elements in *A*, plus one (for one-based indexing). Refer to *rowIndex* array description in [Storage Formats for the Direct Sparse Solvers](#) for more details. The array *ia* is also accessed in all phases of the solution process.

Indexing of *ia* is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter *iparm*(35). For zero-based indexing, the last element *ia*($n+1$) is assumed to be equal to the number of non-zero elements in matrix *A*.

NOTE

For centralized input (*iparm*(40)=0), provide the *ia* array at the master MPI process only. For distributed assembled input (*iparm*(40)=1 or *iparm*(40)=2), provide it at all MPI processes.

ja

INTEGER

Array *ja*(*) contains column indices of the sparse matrix *A* stored in compressed sparse row format. The indices must be stored in increasing order per row. The array *ja* is also accessed in all phases of the solution process. All diagonal elements are stored (even if they are zeros) in the list of non-zero elements in *a* and *ja*. The solver only requires the upper triangular part of the system as is shown for the *columns* array in [Storage Formats for the Direct Sparse Solvers](#).

Indexing of ja is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter [iparm\(35\)](#).

NOTE

For centralized input ([iparm\(40\)=0](#)), provide the ja array at the master MPI process only. For distributed assembled input ([iparm\(40\)=1](#) or [iparm\(40\)=2](#)), provide it at all MPI processes.

perm

INTEGER

Ignored.

nrhs

INTEGER

Number of right-hand sides that need to be solved for.

iparm

INTEGER

Array, dimension (64). This array is used to pass various parameters to Parallel Direct Sparse Solver for Clusters Interface and to return some useful information after execution of the solver.

See [cluster_sparse_solver iparm Parameter](#) for more details about the *iparm* parameters.

msglvl

INTEGER

Message level information. If *msglvl* = 0 then [cluster_sparse_solver](#) generates no output, if *msglvl* = 1 the solver prints statistical information to the screen.

Statistics include information such as the number of non-zero elements in *L-factor* and the timing for each phase.

Set *msglvl* = 1 if you report a problem with the solver, since the additional information provided can facilitate a solution.

b

DOUBLE PRECISION - for real types of matrices (*mtype*=1, 2, -2 and 11) and for double precision Parallel Direct Sparse Solver for Clusters ([iparm\(28\)=0](#))

REAL - for real types of matrices (*mtype*=1, 2, -2 and 11) and for single precision Parallel Direct Sparse Solver for Clusters ([iparm\(28\)=1](#))

DOUBLE COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for double precision Parallel Direct Sparse Solver for Clusters ([iparm\(28\)=0](#))

COMPLEX - for complex types of matrices (*mtype*=3, 6, 13, 14 and -4) and for single precision Parallel Direct Sparse Solver for Clusters ([iparm\(28\)=1](#))

Array, dimension (*n*, *nrhs*). On entry, contains the right-hand side vector/matrix *B*, which is placed in memory contiguously. The *b*(*i*+(*k*-1)×*nrhs*) must hold the *i*-th component of *k*-th right-hand side vector. Note that *b* is only accessed in the solution phase.

comm

INTEGER

MPI communicator. The solver uses the Fortran MPI communicator internally. For C, convert the MPI communicator to Fortran using the `MPI_Comm_c2f()` function. See the examples in the `<install_dir>/examples`

Output Parameters

<i>pt</i>	Handle to internal data structure.												
<i>iparm</i>	On output, some <i>iparm</i> values report information such as the numbers of non-zero elements in the factors. See cluster_sparse_solver iparm Parameter for more details about the <i>iparm</i> parameters.												
<i>b</i>	On output, the array is replaced with the solution if <code>iparm(6) = 1</code> .												
<i>x</i>	DOUBLE PRECISION - for real types of matrices (<i>mtype</i> =1, 2, -2 and 11) and for double precision Parallel Direct Sparse Solver for Clusters (<code>iparm(28)=0</code>) REAL - for real types of matrices (<i>mtype</i> =1, 2, -2 and 11) and for single precision Parallel Direct Sparse Solver for Clusters (<code>iparm(28)=1</code>) DOUBLE COMPLEX - for complex types of matrices (<i>mtype</i> =3, 6, 13, 14 and -4) and for double precision Parallel Direct Sparse Solver for Clusters (<code>iparm(28)=0</code>) COMPLEX - for complex types of matrices (<i>mtype</i> =3, 6, 13, 14 and -4) and for single precision Parallel Direct Sparse Solver for Clusters (<code>iparm(28)=1</code>) Array, dimension (<i>n,nrhs</i>). If <code>iparm(6)=0</code> it contains solution vector/matrix <i>X</i> , which is placed contiguously in memory. The $x(i + (k-1) \times n)$ element must hold the <i>i</i> -th component of the <i>k</i> -th solution vector. Note that <i>x</i> is only accessed in the solution phase.												
<i>error</i>	INTEGER The error indicator according to the below table:												
	<table border="0"> <thead> <tr> <th><i>error</i></th> <th>Information</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>no error</td> </tr> <tr> <td>-1</td> <td>input inconsistent</td> </tr> <tr> <td>-2</td> <td>not enough memory</td> </tr> <tr> <td>-3</td> <td>reordering problem</td> </tr> <tr> <td>-5</td> <td>unclassified (internal) error</td> </tr> </tbody> </table>	<i>error</i>	Information	0	no error	-1	input inconsistent	-2	not enough memory	-3	reordering problem	-5	unclassified (internal) error
<i>error</i>	Information												
0	no error												
-1	input inconsistent												
-2	not enough memory												
-3	reordering problem												
-5	unclassified (internal) error												

cluster_sparse_solver iparm Parameter

The following table describes all individual components of the Parallel Direct Sparse Solver for Clusters Interface `iparm()` parameter. Components which are not used must be initialized with 0. Default values are denoted with an asterisk (*).

Component	Description
<i>iparm</i> (1)	Use default values.
input	0 <i>iparm</i> (2) - <i>iparm</i> (64) are filled with default values. !=0 You must supply all values in components <i>iparm</i> (2) - <i>iparm</i> (64).
<i>iparm</i> (2)	Fill-in reducing ordering for the input matrix.
input	2* The nested dissection algorithm from the METIS package [Karypis98]. 3 The parallel (OpenMP) version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Phase 1 takes significant time.
<i>iparm</i> (3) - <i>iparm</i> (5)	Reserved. Set to zero.
<i>iparm</i> (6)	Write solution on <i>x</i> .
input	<p>NOTE The array <i>x</i> is always used.</p> <hr/> <p>0* The array <i>x</i> contains the solution; right-hand side vector <i>b</i> is kept unchanged.</p> <p>1 The solver stores the solution on the right-hand side <i>b</i>.</p>
<i>iparm</i> (7)	Number of iterative refinement steps performed.
output	Reports the number of iterative refinement steps that were actually performed during the solve step.
<i>iparm</i> (8)	Iterative refinement step.
input	On entry to the solve and iterative refinement step, <i>iparm</i> (8) must be set to the maximum number of iterative refinement steps that the solver performs.
0*	The solver automatically performs two steps of iterative refinement when perturbed pivots are obtained during the numerical factorization.
>0	Maximum number of iterative refinement steps that the solver performs. The solver performs not more than the absolute value of <i>iparm</i> (8) steps of iterative refinement. The solver might stop the process before the maximum number of steps if <ul style="list-style-type: none"> • a satisfactory level of accuracy of the solution in terms of backward error is achieved, • or if it determines that the required accuracy cannot be reached. In this case Parallel Direct Sparse Solver for Clusters Interface returns -4 in the <i>error</i> parameter. The number of executed iterations is reported in <i>iparm</i> (7).
<0	Same as above, but the accumulation of the residuum uses extended precision real and complex data types. Perturbed pivots result in iterative refinement (independent of <i>iparm</i> (8)=0) and the number of executed iterations is reported in <i>iparm</i> (7).
<i>iparm</i> (9)	Reserved. Set to zero.
<i>iparm</i> (10)	Pivoting perturbation.
input	This parameter instructs Parallel Direct Sparse Solver for Clusters Interface how to handle small pivots or zero pivots for nonsymmetric matrices (<i>mtype</i> =11 or <i>mtype</i> =13) and symmetric matrices (<i>mtype</i> =-2, <i>mtype</i> =-4, or <i>mtype</i> =6). For these matrices the solver

Component	Description
	<p>uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04].</p> <p>Small pivots are perturbed with $\text{eps} = 10^{-\text{iparm}(10)}$.</p> <p>The magnitude of the potential pivot is tested against a constant threshold of $\text{alpha} = \text{eps} * \ A_2\ _{\infty}$, where $\text{eps} = 10^{(-\text{iparm}(10))}$, $A_2 = P^* P_{MPS}^* D_r^* A^* D_c^* P$, and $\ A_2\ _{\infty}$ is the infinity norm of the scaled and permuted matrix A. Any tiny pivots encountered during elimination are set to the sign $(\text{L}_1)^* \text{eps} * \ A_2\ _{\infty}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with $\text{eps} = 10^{(-\text{iparm}(10))}$.</p>
$\text{iparm}(11)$	<p>Scaling vectors.</p> <p>input</p> <p>Parallel Direct Sparse Solver for Clusters Interface uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale.</p> <p>Use $\text{iparm}(11) = 1$ (scaling) and $\text{iparm}(13) = 1$ (matching) for highly indefinite symmetric matrices, for example, from interior point optimizations or saddle point problems. Note that in the analysis phase ($\text{phase}=11$) you must provide the numerical values of the matrix A in array a in case of scaling and symmetric weighted matching.</p>
	<p>0* Disable scaling. Default for symmetric indefinite matrices.</p> <p>1* Enable scaling. Default for nonsymmetric matrices.</p> <p>Scale the matrix so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less or equal to 1. This scaling method is applied to nonsymmetric matrices ($\text{mtype} = 11, \text{mtype} = 13$). The scaling can also be used for symmetric indefinite matrices ($\text{mtype} = -2, \text{mtype} = -4, \text{mtype} = 6$) when the symmetric weighted matchings are applied ($\text{iparm}(13) = 1$).</p> <p>Note that in the analysis phase ($\text{phase}=11$) you must provide the numerical values of the matrix A in case of scaling.</p>
$\text{iparm}(12)$	Reserved. Set to zero.
$\text{iparm}(13)$	Improved accuracy using (non-) symmetric weighted matching.
input	<p>Parallel Direct Sparse Solver for Clusters Interface can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to the factorization methods and complements the alternative of using more complete pivoting techniques during the numerical factorization.</p>
	<p>0* Disable matching. Default for symmetric indefinite matrices.</p> <p>1* Enable matching. Default for nonsymmetric matrices.</p> <p>Maximum weighted matching algorithm to permute large elements close to the diagonal.</p> <p>It is recommended to use $\text{iparm}(11) = 1$ (scaling) and $\text{iparm}(13) = 1$ (matching) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems.</p>

Component	Description				
	Note that in the analysis phase (<i>phase</i> =11) you must provide the numerical values of the matrix <i>A</i> in case of symmetric weighted matching.				
<i>iparm</i> (14)	Reserved. Set to zero.				
-					
<i>iparm</i> (20)					
<i>iparm</i> (21)	Pivoting for symmetric indefinite matrices.				
input	<table> <tr> <td>0</td><td>Apply 1x1 diagonal pivoting during the factorization process.</td></tr> <tr> <td>1*</td><td>Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <i>mtype</i>=-2, <i>mtype</i>=-4, or <i>mtype</i>=6.</td></tr> </table>	0	Apply 1x1 diagonal pivoting during the factorization process.	1*	Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <i>mtype</i> =-2, <i>mtype</i> =-4, or <i>mtype</i> =6.
0	Apply 1x1 diagonal pivoting during the factorization process.				
1*	Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of <i>mtype</i> =-2, <i>mtype</i> =-4, or <i>mtype</i> =6.				
<i>iparm</i> (22)	Reserved. Set to zero.				
-					
<i>iparm</i> (26)					
<i>iparm</i> (27)	Matrix checker.				
input	<table> <tr> <td>0*</td><td>Do not check the sparse matrix representation for errors.</td></tr> <tr> <td>1</td><td>Check integer arrays <i>ia</i> and <i>ja</i>. In particular, check whether the column indices are sorted in increasing order within each row.</td></tr> </table>	0*	Do not check the sparse matrix representation for errors.	1	Check integer arrays <i>ia</i> and <i>ja</i> . In particular, check whether the column indices are sorted in increasing order within each row.
0*	Do not check the sparse matrix representation for errors.				
1	Check integer arrays <i>ia</i> and <i>ja</i> . In particular, check whether the column indices are sorted in increasing order within each row.				
<i>iparm</i> (28)	Single or double precision Parallel Direct Sparse Solver for Clusters Interface.				
input	<p>See iparm(8) for information on controlling the precision of the refinement steps.</p> <table> <tr> <td>0*</td><td>Input arrays (<i>a</i>, <i>x</i> and <i>b</i>) and all internal arrays must be presented in double precision.</td></tr> <tr> <td>1</td><td>Input arrays (<i>a</i>, <i>x</i> and <i>b</i>) must be presented in single precision. In this case all internal computations are performed in single precision.</td></tr> </table>	0*	Input arrays (<i>a</i> , <i>x</i> and <i>b</i>) and all internal arrays must be presented in double precision.	1	Input arrays (<i>a</i> , <i>x</i> and <i>b</i>) must be presented in single precision. In this case all internal computations are performed in single precision.
0*	Input arrays (<i>a</i> , <i>x</i> and <i>b</i>) and all internal arrays must be presented in double precision.				
1	Input arrays (<i>a</i> , <i>x</i> and <i>b</i>) must be presented in single precision. In this case all internal computations are performed in single precision.				
<i>iparm</i> (29)	Reserved. Set to zero.				
-					
<i>iparm</i> (34)					
<i>iparm</i> (35)	One- or zero-based indexing of columns and rows.				
input	<table> <tr> <td>0*</td><td>One-based indexing: columns and rows indexing in arrays <i>ia</i>, <i>ja</i>, and <i>perm</i> starts from 1 (Fortran-style indexing).</td></tr> <tr> <td>1</td><td>Zero-based indexing: columns and rows indexing in arrays <i>ia</i>, <i>ja</i>, and <i>perm</i> starts from 0 (C-style indexing).</td></tr> </table>	0*	One-based indexing: columns and rows indexing in arrays <i>ia</i> , <i>ja</i> , and <i>perm</i> starts from 1 (Fortran-style indexing).	1	Zero-based indexing: columns and rows indexing in arrays <i>ia</i> , <i>ja</i> , and <i>perm</i> starts from 0 (C-style indexing).
0*	One-based indexing: columns and rows indexing in arrays <i>ia</i> , <i>ja</i> , and <i>perm</i> starts from 1 (Fortran-style indexing).				
1	Zero-based indexing: columns and rows indexing in arrays <i>ia</i> , <i>ja</i> , and <i>perm</i> starts from 0 (C-style indexing).				
<i>iparm</i> (36)	Reserved. Set to zero.				
-					
<i>iparm</i> (39)					
<i>iparm</i> (40)	Matrix input format.				
input	<p>NOTE</p> <p>Performance of the reordering step of the Parallel Direct Sparse Solver for Clusters Interface is slightly better for assembled format (CSR, <i>iparm</i>(40) = 0) than for distributed format (DCSR, <i>iparm</i>(40) > 0) for the same matrices, so if the matrix is assembled on one node do not distribute it before calling <code>cluster_sparse_solver</code>.</p>				
0*	Provide the matrix in usual centralized input format: the master MPI process stores all data from matrix <i>A</i> , with rank=0.				

Component	Description
1	Provide the matrix in distributed assembled matrix input format. In this case, each MPI process stores only a part (or domain) of the matrix A data. Set the bounds of the domain using <code>iparm(41)</code> and <code>iparm(42)</code> . The solution vector is placed on the master process.
2	Provide the matrix in distributed assembled matrix input format. In this case, each MPI process stores only a part (or domain) of the matrix A data. Set the bounds of the domain using <code>iparm(41)</code> and <code>iparm(42)</code> . The solution vector, A , and RHS elements are distributed between processes in same manner.
<code>iparm(41)</code>	Beginning of input domain.
input	The number of the matrix A row, RHS element, and, for $\text{iparm}(40)=2$, solution vector that begins the input domain belonging to this MPI process. Only applicable to the distributed assembled matrix input format ($\text{iparm}(40)=1$ or $\text{iparm}(40)=2$). See Sparse Matrix Storage Formats for more details.
<code>iparm(42)</code>	End of input domain.
input	The number of the matrix A row, RHS element, and, for $\text{iparm}(40)=2$, solution vector that ends the input domain belonging to this MPI process. Only applicable to the distributed assembled matrix input format ($\text{iparm}(40)=1$ or $\text{iparm}(40)=2$). See Sparse Matrix Storage Formats for more details.
<code>iparm(43)</code>	Reserved. Set to zero.
-	
<code>iparm(64)</code>	
input	

NOTE

Generally in sparse matrices, components which are equal to zero can be considered non-zero if necessary. For example, in order to make a matrix structurally symmetric, elements which are zero can be considered non-zero. See [Sparse Matrix Storage Formats](#) for an example.

Direct Sparse Solver (DSS) Interface Routines

Intel MKL supports the DSS interface, an alternative to the Intel MKL PARDISO interface for the direct sparse solver. The DSS interface implements a group of user-callable routines that are used in the step-by-step solving process and utilizes the general scheme described in [Appendix A Linear Solvers Basics](#) for solving sparse systems of linear equations. This interface also includes one routine for gathering statistics related to the solving process and an auxiliary routine for passing character strings from Fortran routines to C routines.

The DSS interface also supports the out-of-core (OOC) mode.

Table "DSS Interface Routines" lists the names of the routines and describes their general use.

DSS Interface Routines

Routine	Description
<code>dss_create</code>	Initializes the solver and creates the basic data structures necessary for the solver. This routine must be called before any other DSS routine.

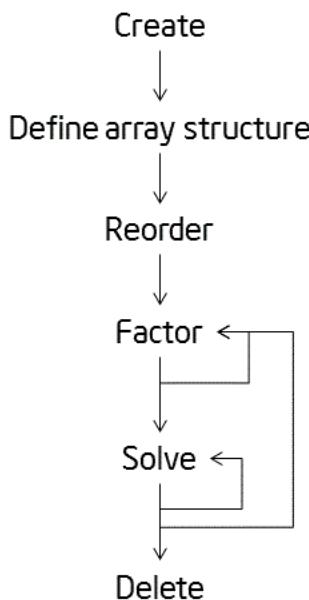
Routine	Description
<code>dss_define_structure</code>	Informs the solver of the locations of the non-zero elements of the matrix.
<code>dss_reorder</code>	Based on the non-zero structure of the matrix, computes a permutation vector to reduce fill-in during the factoring process.
<code>dss_factor_real, dss_factor_complex</code>	Computes the LU , LDL^T or LL^T factorization of a real or complex matrix.
<code>dss_solve_real, dss_solve_complex</code>	Computes the solution vector for a system of equations based on the factorization computed in the previous phase.
<code>dss_delete</code>	Deletes all data structures created during the solving process.
<code>dss_statistics</code>	Returns statistics about various phases of the solving process.
<code>mkl_cvt_to_null_terminated_str</code>	Passes character strings from Fortran routines to C routines.

To find a single solution vector for a single system of equations with a single right-hand side, invoke the Intel MKL DSS interface routines in this order:

1. `dss_create`
2. `dss_define_structure`
3. `dss_reorder`
4. `dss_factor_real, dss_factor_complex`
5. `dss_solve_real, dss_solve_complex`
6. `dss_delete`

However, in certain applications it is necessary to produce solution vectors for multiple right-hand sides for a given factorization and/or factor several matrices with the same non-zero structure. Consequently, it is sometimes necessary to invoke the Intel MKL sparse routines in an order other than that listed, which is possible using the DSS interface. The solving process is conceptually divided into six phases. [Figure "Typical order for invoking DSS interface routines"](#) indicates the typical order in which the DSS interface routines can be invoked.

[Typical order for invoking DSS interface routines](#)



See the code examples that use the DSS interface routines to solve systems of linear equations in the Intel MKL installation directory (`dss_*.f` and `dss_*.c`).

- Fortran examples: `examples/solverf/source`
- C examples: `examples/solverc/source`

DSS Interface Description

Each DSS routine reads from or writes to a data object called a *handle*. Refer to [Memory Allocation and Handles](#) to determine the correct method for declaring a handle argument for each language. For simplicity, the descriptions in DSS routines refer to the data type as `MKL_DSS_HANDLE`.

C and C++ programmers should refer to [Calling Sparse Solver and Preconditioner Routines from C/C++](#) for information on mapping Fortran types to C/C++ types.

Routine Options

The DSS routines have an integer argument (referred below to as *opt*) for passing various options to the routines. The permissible values for *opt* should be specified using only the symbol constants defined in the language-specific header files (see [Implementation Details](#)). The routines accept options for setting the message and termination levels as described in [Table "Symbolic Names for the Message and Termination Levels Options"](#). Additionally, each routine accepts the option `MKL_DSS_DEFAULTS` that sets the default values (as documented) for *opt* to the routine.

Symbolic Names for the Message and Termination Levels Options

Message Level	Termination Level
<code>MKL_DSS_MSG_LVL_SUCCESS</code>	<code>MKL_DSS_TERM_LVL_SUCCESS</code>
<code>MKL_DSS_MSG_LVL_INFO</code>	<code>MKL_DSS_TERM_LVL_INFO</code>
<code>MKL_DSS_MSG_LVL_WARNING</code>	<code>MKL_DSS_TERM_LVL_WARNING</code>
<code>MKL_DSS_MSG_LVL_ERROR</code>	<code>MKL_DSS_TERM_LVL_ERROR</code>
<code>MKL_DSS_MSG_LVL_FATAL</code>	<code>MKL_DSS_TERM_LVL_FATAL</code>

The settings for message and termination levels can be set on any call to a DSS routine. However, once set to a particular level, they remain at that level until they are changed in another call to a DSS routine.

You can specify both message and termination level for a DSS routine by adding the options together. For example, to set the message level to `debug` and the termination level to `error` for all the DSS routines, use the following call:

```
CALL dss_create( handle, MKL_DSS_MSG_LVL_INFO + MKL_DSS_TERM_LVL_ERROR)
```

User Data Arrays

Many of the DSS routines take arrays of user data as input. For example, user arrays are passed to the routine `dss_define_structure` to describe the location of the non-zero entries in the matrix.

CAUTION

Do not modify the contents of these arrays after they are passed to one of the solver routines.

DSS Implementation Details

To promote portability across platforms and ease of use across different languages, one of the following Intel MKL DSS language-specific header files can be included:

- `mkl_dss.f77` for F77 programs
- `mkl_dss.f90` for F90 programs
- `mkl_dss.h` for C programs

These header files define symbolic constants for returned error values, function options, certain defined data types, and function prototypes.

NOTE

Constants for options, returned error values, and message severities must be referred only by the symbolic names that are defined in these header files. Use of the Intel MKL DSS software without including one of the above header files is not supported.

Memory Allocation and Handles

You do not need to allocate any temporary working storage in order to use the Intel MKL DSS routines, because the solver itself allocates any required storage. To enable multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using a *handle* data object.

Each of the Intel MKL DSS routines creates, uses, or deletes a handle. Consequently, any program calling an Intel MKL DSS routine must be able to allocate storage for a handle. The exact syntax for allocating storage for a handle varies from language to language. To standardize the handle declarations, the language-specific header files declare constants and defined data types that must be used when declaring a handle object in your code.

- C and C++:

```
#include "mkl_dss.h"
_MKL_DSS_HANDLE_t handle;
```

- Fortran 90:

```
INCLUDE "mkl_dss.f90"
TYPE(MKL_DSS_HANDLE) handle
```

- FORTRAN 77 (compilers that support eight byte integers):

```
INCLUDE "mkl_dss.f77"
INTEGER*8 handle
```

In addition to the definition for the correct declaration of a handle, the include file also defines the following:

- function prototypes for languages that support prototypes
- symbolic constants that are used for the returned error values
- user options for the solver routines
- constants indicating message severity.

DSS Routines

dss_create

Initializes the solver.

Syntax

Fortran:

```
call dss_create(handle, opt)
```

C:

```
MKL_INT dss_create(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt)
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_dss.f90
- C: mkl.h

Description

The `dss_create` routine initializes the solver. After the call to `dss_create`, all subsequent invocations of the Intel MKL DSS routines must use the value of the handle returned by `dss_create`.

WARNING

Do not write the value of handle directly.

The default value of the parameter `opt` is

```
MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR.
```

By default, the DSS routines use double precision for solving systems of linear equations. The precision used by the DSS routines can be set to single mode by adding the following value to the `opt` parameter:

```
MKL_DSS_SINGLE_PRECISION.
```

Input data and internal arrays are required to have single precision.

By default, the DSS routines use Fortran style indexing for input arrays of integer types (the first value is referenced as array element 1), regardless of whether the Fortran or the C interface is used. To set indexing to C style (the first value is referenced as array element 0), add the following value to the `opt` parameter:

```
MKL_DSS_ZERO_BASED_INDEXING.
```

The `opt` parameter can also control number of refinement steps used on the solution stage by specifying the two following values:

`MKL_DSS_REFINEMENT_OFF` - maximum number of refinement steps is set to zero;

`MKL_DSS_REFINEMENT_ON` (default value) - maximum number of refinement steps is set to 2.

By default, DSS uses in-core computations. To launch the out-of-core version of DSS (OOC DSS) you can add to this parameter one of two possible values: `MKL_DSS_OOC_STRONG` and `MKL_DSS_OOC_VARIABLE`.

`MKL_DSS_OOC_STRONG` - OOC DSS is used.

`MKL_DSS_OOC_VARIABLE` - if the memory needed for the matrix factors is less than the value of the environment variable `MKL_PARDISO_OOC_MAX_CORE_SIZE`, then the OOC DSS uses the in-core kernels of Intel MKL PARDISO, otherwise it uses the OOC computations.

The variable `MKL_PARDISO_OOC_MAX_CORE_SIZE` defines the maximum size of RAM allowed for storing work arrays associated with the matrix factors. It is ignored if `MKL_DSS_OOC_STRONG` is set. The default value of `MKL_PARDISO_OOC_MAX_CORE_SIZE` is 2000 MB. This value and default path and file name for storing temporary data can be changed using the configuration file `pardiso_ooc.cfg` or command line (See more details in the description of the [pardiso](#) routine).

WARNING

Other than message and termination level options, do not change the OOC DSS settings after they are specified in the routine `dss_create`.

Input Parameters

opt

FORTRAN 77: INTEGER

Fortran 90: INTEGER, INTENT (IN)

Parameter to pass the DSS options. The default value is
`MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR`.

Output Parameters

handle

FORTRAN 77: INTEGER*8

Fortran 90: TYPE (`MKL_DSS_HANDLE`), INTENT (OUT)

Pointer to the data structure storing internal DSS results
`(MKL_DSS_HANDLE)`.

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_INVALID_OPTION`

`MKL_DSS_OUT_OF_MEMORY`

`MKL_DSS_MSG_LVL_ERR`

`MKL_DSS_TERM_LVL_ERR`

[dss_define_structure](#)

Communicates locations of non-zero elements in the matrix to the solver.

Syntax

Fortran:

```
call dss_define_structure(handle, opt, rowIndex, nRows, nCols, columns, nNonZeros);
```

C:

```
MKL_INT dss_define_structure(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt,
MKL_INT const *rowIndex, MKL_INT const *nRows, MKL_INT const *nCols,
MKL_INT const *columns, MKL_INT const *nNonZeros);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_dss.f90
- C: mkl.h

Description

The routine `dss_define_structure` communicates the locations of the `nNonZeros` number of non-zero elements in a matrix of `nRows * nCols` size to the solver.

NOTE

The Intel MKL DSS software operates only on square matrices, so `nRows` must be equal to `nCols`.

To communicate the locations of non-zero elements in the matrix, do the following:

1. Define the general non-zero structure of the matrix by specifying the value for the options argument `opt`. You can set the following values for real matrices:
 - `MKL_DSS_SYMMETRIC_STRUCTURE`
 - `MKL_DSS_SYMMETRIC`
 - `MKL_DSS_NON_SYMMETRIC`
 and for complex matrices:
 - `MKL_DSS_SYMMETRIC_STRUCTURE_COMPLEX`
 - `MKL_DSS_SYMMETRIC_COMPLEX`
 - `MKL_DSS_NON_SYMMETRIC_COMPLEX`
 The information about the matrix type must be defined in `dss_define_structure`.
2. Provide the actual locations of the non-zeros by means of the arrays `rowIndex` and `columns` (see [Sparse Matrix Storage Format](#)).

Input Parameters

`opt`

FORTRAN 77: INTEGER

Fortran 90: INTEGER, INTENT (IN)

Parameter to pass the DSS options. The default value for the matrix structure is `MKL_DSS_SYMMETRIC`.

`rowIndex`

FORTRAN 77: INTEGER

Fortran 90: INTEGER, INTENT (IN)

Array of size `nRows+1`. Defines the location of non-zero entries in the matrix.

`nRows`

FORTRAN 77: INTEGER

Fortran 90: INTEGER, INTENT (IN)

Number of rows in the matrix.

<i>nCols</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN)
Number of columns in the matrix; must be equal to <i>nRows</i> .	
<i>columns</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN)
Array of size <i>nNonZeros</i> . Defines the column location of non-zero entries in the matrix.	
<i>nNonZeros</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN)
Number of non-zero elements in the matrix.	

Output Parameters

<i>handle</i>	FORTRAN 77: INTEGER*8 Fortran 90: TYPE (MKL_DSS_HANDLE), INTENT (INOUT)
Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).	

Return Values

MKL_DSS_SUCCESS
MKL_DSS_STATE_ERR
MKL_DSS_INVALID_OPTION
MKL_DSS_STRUCTURE_ERR
MKL_DSS_ROW_ERR
MKL_DSS_COL_ERR
MKL_DSS_NOT_SQUARE
MKL_DSS_TOO_FEW_VALUES
MKL_DSS_TOO_MANY_VALUES
MKL_DSS_OUT_OF_MEMORY
MKL_DSS_MSG_LVL_ERR
MKL_DSS_TERM_LVL_ERR

dss_reordered

Computes or sets a permutation vector that minimizes the fill-in during the factorization phase.

Syntax

Fortran:

```
call dss_reordered(handle, opt, perm)
```

C:

```
MKL_INT dss_reordered(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, MKL_INT const *perm)
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_dss.f90
- C: mkl.h

Description

If *opt* contains the option MKL_DSS_AUTO_ORDER, then the routine dss_reorder computes a permutation vector that minimizes the fill-in during the factorization phase. For this option, the routine ignores the contents of the *perm* array.

If *opt* contains the option MKL_DSS_METIS_OPENMP_ORDER, then the routine dss_reorder computes permutation vector using the parallel nested dissections algorithm to minimize the fill-in during the factorization phase. This option can be used to decrease the time of dss_reorder call on multi-core computers. For this option, the routine ignores the contents of the *perm* array.

If *opt* contains the option MKL_DSS_MY_ORDER, then you must supply a permutation vector in the array *perm*. In this case, the array *perm* is of length *nRows*, where *nRows* is the number of rows in the matrix as defined by the previous call to [dss_define_structure](#).

If *opt* contains the option MKL_DSS_GET_ORDER, then the permutation vector computed during the dss_reorder call is copied to the array *perm*. In this case you must allocate the array *perm* beforehand. The permutation vector is computed in the same way as if the option MKL_DSS_AUTO_ORDER is set.

Input Parameters

<i>opt</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN)
Parameter to pass the DSS options. The default value for the permutation type is MKL_DSS_AUTO_ORDER.	
<i>perm</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN)
Array of length <i>nRows</i> . Contains a user-defined permutation vector (accessed only if <i>opt</i> contains MKL_DSS_MY_ORDER or MKL_DSS_GET_ORDER).	

Output Parameters

<i>handle</i>	FORTRAN 77: INTEGER*8 Fortran 90: TYPE (MKL_DSS_HANDLE), INTENT(INOUT)
Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).	

Return Values

MKL_DSS_SUCCESS
MKL_DSS_STATE_ERR
MKL_DSS_INVALID_OPTION
MKL_DSS_REORDER_ERR
MKL_DSS_REORDER1_ERR
MKL_DSS_I32BIT_ERR

```
MKL_DSS_FAILURE
MKL_DSS_OUT_OF_MEMORY
MKL_DSS_MSG_LVL_ERR
MKL_DSS_TERM_LVL_ERR
```

dss_factor_real, dss_factor_complex

Compute factorization of the matrix with previously specified location of non-zero elements.

Syntax

FORTRAN 77:

```
call dss_factor_real(handle, opt, rValues)
call dss_factor_complex(handle, opt, cValues)
```

Fortran 90:

```
call dss_factor(handle, opt, Values)
```

C:

```
MKL_INT dss_factor_real(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt,
void const *rValues)

MKL_INT dss_factor_complex(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt,
void const *cValues)
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_dss.f90
- C: mkl.h

Description

These routines compute factorization of the matrix whose non-zero locations were previously specified by a call to [dss_define_structure](#) and whose non-zero values are given in the array *rValues*, *cValues* or *Values*. Data type These arrays must be of length *nNonZeros* as defined in a previous call to *dss_define_structure*.

NOTE

The data type (single or double precision) of *rValues*, *cValues*, *Values* must be in correspondence with precision specified by the parameter *opt* in the routine *dss_create*.

The *opt* argument can contain one of the following options:

- MKL_DSS_POSITIVE_DEFINITE
- MKL_DSS_INDEFINITE
- MKL_DSS_HERMITIAN_POSITIVE_DEFINITE
- MKL_DSS_HERMITIAN_INDEFINITE

depending on your matrix's type.

NOTE

This routine supports the Progress Routine feature. See [Progress Function](#) section for details.

Input Parameters

<i>handle</i>	FORTRAN 77: INTEGER*8 Fortran 90: TYPE (MKL_DSS_HANDLE), INTENT(INOUT)
	Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).
<i>opt</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN)
	Parameter to pass the DSS options. The default value is MKL_DSS_POSITIVE_DEFINITE.
<i>rValues</i>	FORTRAN 77: REAL*4 or REAL*8 Fortran 90: REAL(KIND=4), INTENT(IN) or REAL(KIND=8), INTENT(IN)
	Array of elements of the matrix <i>A</i> . Real data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <code>dss_create</code> .
<i>cValues</i>	FORTRAN 77: COMPLEX*8 or COMPLEX*16 Fortran 90: COMPLEX(KIND=4), INTENT(IN) or COMPLEX(KIND=8), INTENT(IN)
	Array of elements of the matrix <i>A</i> . Complex data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <code>dss_create</code> .
<i>Values</i>	 Fortran 90: REAL(KIND=4), INTENT(OUT), or REAL(KIND=8), INTENT(OUT), or COMPLEX(KIND=4), INTENT(OUT), or COMPLEX(KIND=8), INTENT(OUT)
	Array of elements of the matrix <i>A</i> . Real or complex data, single or double precision as it is specified by the parameter <i>opt</i> in the routine <code>dss_create</code> .

Return Values

MKL_DSS_SUCCESS

MKL_DSS_STATE_ERR

MKL_DSS_INVALID_OPTION

MKL_DSS_OPTION_CONFLICT

MKL_DSS_VALUES_ERR

MKL_DSS_OUT_OF_MEMORY

```
MKL_DSS_ZERO_PIVOT
MKL_DSS_FAILURE
MKL_DSS_MSG_LVL_ERR
MKL_DSS_TERM_LVL_ERR
MKL_DSS_OOC_MEM_ERR
MKL_DSS_OOC_OC_ERR
MKL_DSS_OOC_RW_ERR
```

[dss_solve_real, dss_solve_complex](#)

Compute the corresponding solution vector and place it in the output array.

Syntax

FORTRAN 77:

```
call dss_solve_real(handle, opt, rRhsValues, nRhs, rSolValues)
call dss_solve_complex(handle, opt, cRhsValues, nRhs, cSolValues)
```

Fortran 90:

```
call dss_solve(handle, opt, RhsValues, nRhs, SolValues)
```

C:

```
MKL_INT dss_solve_real(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt,
void const *rRhsValues, MKL_INT const *nRhs, void *rSolValues)

MKL_INT dss_solve_complex(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt,
void const *cRhsValues, MKL_INT const *nRhs, void *cSolValues)
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_dss.f90
- C: mkl.h

Description

For each right-hand side column vector defined in the arrays *rRhsValues*, *cRhsValues*, or *RhsValues*, these routines compute the corresponding solution vector and place it in the arrays *rSolValues*, *cSolValues*, or *SolValues* respectively.

NOTE

The data type (single or double precision) of all arrays must be in correspondence with precision specified by the parameter *opt* in the routine *dss_create*.

The lengths of the right-hand side and solution vectors, *nCols* and *nRows* respectively, must be defined in a previous call to [dss_define_structure](#).

By default, both routines perform the full solution step (it corresponds to *phase* = 33 in Intel MKL PARDISO). The parameter *opt* enables you to calculate the final solution step-by-step, calling forward and backward substitutions.

If it is set to *MKL_DSS_FORWARD_SOLVE*, the forward substitution (corresponding to *phase* = 331 in Intel MKL PARDISO) is performed;

if it is set to `MKL_DSS_DIAGONAL_SOLVE`, the diagonal substitution (corresponding to `phase = 332` in Intel MKL PARDISO) is performed, if possible;

if it is set to `MKL_DSS_BACKWARD_SOLVE`, the backward substitution (corresponding to `phase = 333` in Intel MKL PARDISO) is performed.

For more details about using these substitutions for different types of matrices, see [Separate Forward and Backward Substitution](#) in the Intel MKL PARDISO solver description.

This parameter also can control the number of refinement steps that is used on the solution stage: if it is set to `MKL_DSS_REFINEMENT_OFF`, the maximum number of refinement steps equal to zero, and if it is set to `MKL_DSS_REFINEMENT_ON` (default value), the maximum number of refinement steps is equal to 2.

`MKL_DSS_CONJUGATE_SOLVE` option added to the parameter `opt` enables solving a conjugate transposed system $A^H X = B$ based on the factorization of the matrix A . This option is equivalent to the parameter `iparm(12) = 1` in Intel MKL PARDISO.

`MKL_DSS_TRANSPOSE_SOLVE` option added to the parameter `opt` enables solving a transposed system $A^T X = B$ based on the factorization of the matrix A . This option is equivalent to the parameter `iparm(12) = 2` in Intel MKL PARDISO.

Input Parameters

`handle`

FORTRAN 77: INTEGER*8

Fortran 90: TYPE (`MKL_DSS_HANDLE`), INTENT (INOUT)

Pointer to the data structure storing internal DSS results (`MKL_DSS_HANDLE`).

`opt`

FORTRAN 77: INTEGER

Fortran 90: INTEGER, INTENT (IN)

Parameter to pass the DSS options.

`nRhs`

FORTRAN 77: INTEGER

Fortran 90: INTEGER, INTENT (IN)

Number of the right-hand sides in the system of linear equations.

`rRhsValues`

FORTRAN 77: REAL*4 or

REAL*8

Fortran 90: REAL (KIND=4), INTENT (IN) or

REAL (KIND=8), INTENT (IN)

Array of size `nRows * nRhs`. Contains real right-hand side vectors. Real data, single or double precision as it is specified by the parameter `opt` in the routine `dss_create`.

`cRhsValues`

FORTRAN 77: COMPLEX*8 or

COMPLEX*16

Fortran 90: COMPLEX (KIND=4), INTENT (IN) or

COMPLEX (KIND=8), INTENT (IN)

Array of size `nRows * nRhs`. Contains complex right-hand side vectors. Complex data, single or double precision as it is specified by the parameter `opt` in the routine `dss_create`.

RhsValues

Fortran 90: REAL (KIND=4), INTENT (IN), or
 REAL (KIND=8), INTENT (IN), or
 COMPLEX (KIND=4), INTENT (IN), or
 COMPLEX (KIND=8), INTENT (IN)
 Array of size $nRows * nRhs$. Contains right-hand side vectors. Real or complex data, single or double precision as it is specified by the parameter opt in the routine `dss_create`.

Output Parameters

rSolValues

FORTRAN 77: REAL*4 or
 REAL*8
Fortran 90: REAL (KIND=4), INTENT (OUT) or
 REAL (KIND=8), INTENT (OUT)
 Array of size $nCols * nRhs$. Contains real solution vectors. Real data, single or double precision as it is specified by the parameter opt in the routine `dss_create`.

cSolValues

FORTRAN 77: COMPLEX*8 or
 COMPLEX*16
Fortran 90: COMPLEX (KIND=4), INTENT (OUT) or
 COMPLEX (KIND=8), INTENT (OUT)
 Array of size $nCols * nRhs$. Contains complex solution vectors. Complex data, single or double precision as it is specified by the parameter opt in the routine `dss_create`.

SolValues

Fortran 90: REAL (KIND=4), INTENT (OUT), or
 REAL (KIND=8), INTENT (OUT), or
 COMPLEX (KIND=4), INTENT (OUT), or
 COMPLEX (KIND=8), INTENT (OUT)
 Array of size $nCols * nRhs$. Contains solution vectors. Real or complex data, single or double precision as it is specified by the parameter opt in the routine `dss_create`.

Return Values

MKL_DSS_SUCCESS
 MKL_DSS_STATE_ERR
 MKL_DSS_INVALID_OPTION
 MKL_DSS_OUT_OF_MEMORY
 MKL_DSS_DIAG_ERR
 MKL_DSS_FAILURE
 MKL_DSS_MSG_LVL_ERR

```
MKL_DSS_TERM_LVL_ERR  
MKL_DSS_OOC_MEM_ERR  
MKL_DSS_OOC_OC_ERR  
MKL_DSS_OOC_RW_ERR
```

dss_delete

Deletes all of the data structures created during the solutions process.

Syntax

Fortran:

```
call dss_delete(handle, opt)
```

C:

```
MKL_INT dss_delete(_MKL_DSS_HANDLE_t const *handle, MKL_INT const *opt)
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_dss.f90
- C: mkl.h

Description

The routine `dss_delete` deletes all data structures created during the solving process.

Input Parameters

opt

FORTRAN 77: INTEGER

Fortran 90: INTEGER, INTENT(IN)

Parameter to pass the DSS options. The default value is
`MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR.`

Output Parameters

handle

FORTRAN 77: INTEGER*8

Fortran 90: TYPE (MKL_DSS_HANDLE), INTENT(INOUT)

Pointer to the data structure storing internal DSS results
(`MKL_DSS_HANDLE`).

Return Values

```
MKL_DSS_SUCCESS  
MKL_DSS_STATE_ERR  
MKL_DSS_INVALID_OPTION  
MKL_DSS_OUT_OF_MEMORY  
MKL_DSS_MSG_LVL_ERR  
MKL_DSS_TERM_LVL_ERR
```

dss_statistics

Returns statistics about various phases of the solving process.

Syntax

Fortran:

```
call dss_statistics(handle, opt, statArr, retValues)
```

C:

```
MKL_INT dss_statistics(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt,
_CHARACTER_STR_t const *statArr, _DOUBLE_PRECISION_t *retValues)
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_dss.f90
- C: mkl.h

Description

The dss_statistics routine returns statistics about various phases of the solving process. This routine gathers the following statistics:

- time taken to do reordering,
- time taken to do factorization,
- duration of problem solving,
- determinant of the symmetric indefinite input matrix,
- inertia of the symmetric indefinite input matrix,
- number of floating point operations taken during factorization,
- total peak memory needed during the analysis and symbolic factorization,
- permanent memory needed from the analysis and symbolic factorization,
- memory consumption for the factorization and solve phases.

Statistics are returned in accordance with the input string specified by the parameter *statArr*. The value of the statistics is returned in double precision in a return array, which you must allocate beforehand.

For multiple statistics, multiple string constants separated by commas can be used as input. Return values are put into the return array in the same order as specified in the input string.

Statistics can only be requested at the appropriate stages of the solving process. For example, requesting FactorTime before a matrix is factored leads to an error.

The following table shows the point at which each individual statistics item can be requested:

Statistics Calling Sequences

Type of Statistics	When to Call
ReorderTime	After dss_reorder is completed successfully.
FactorTime	After dss_factor_real or dss_factor_complex is completed successfully.
SolveTime	After dss_solve_real or dss_solve_complex is completed successfully.
Determinant	After dss_factor_real or dss_factor_complex is completed successfully.
Inertia	After dss_factor_real is completed successfully and the matrix is real, symmetric, and indefinite.
Flops	After dss_factor_real or dss_factor_complex is completed successfully.
Peakmem	After dss_reorder is completed successfully.

Type of Statistics	When to Call
Factormem	After <code>dss_reordered</code> is completed successfully.
Solvemem	After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.

Input Parameters

<i>handle</i>	FORTRAN 77: INTEGER*8 Fortran 90: TYPE (MKL_DSS_HANDLE), INTENT(IN) Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).										
<i>opt</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) Parameter to pass the DSS options.										
<i>statArr</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) Input string that defines the type of the returned statistics. The parameter can include one or more of the following string constants (case of the input string has no effect): <table> <tbody> <tr> <td>ReorderTime</td><td>Amount of time taken to do the reordering.</td></tr> <tr> <td>FactorTime</td><td>Amount of time taken to do the factorization.</td></tr> <tr> <td>SolveTime</td><td>Amount of time taken to solve the problem after factorization.</td></tr> <tr> <td>Determinant</td><td> Determinant of the matrix A. For real matrices: the determinant is returned as <code>det_pow</code>, <code>det_base</code> in two consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_base}) < 10.0$ and <code>determinant = det_base * 10^(det_pow)</code>. For complex matrices: the determinant is returned as <code>det_pow</code>, <code>det_re</code>, <code>det_im</code> in three consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_re}) + \text{abs}(\text{det_im}) < 10.0$ and <code>determinant = (det_re, det_im) * 10^(det_pow)</code>. </td></tr> <tr> <td>Inertia</td><td> Inertia of a real symmetric matrix is defined as a triplet of nonnegative integers (p, n, z), where p is the number of positive eigenvalues, n is the number of negative eigenvalues, and z is the number of zero eigenvalues. Inertia is returned as three consecutive return array locations p, n, z. Computing inertia can lead to incorrect results for matrixes with a cluster of eigenvalues which are near 0. </td></tr> </tbody> </table>	ReorderTime	Amount of time taken to do the reordering.	FactorTime	Amount of time taken to do the factorization.	SolveTime	Amount of time taken to solve the problem after factorization.	Determinant	Determinant of the matrix A . For real matrices: the determinant is returned as <code>det_pow</code> , <code>det_base</code> in two consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_base}) < 10.0$ and <code>determinant = det_base * 10^(det_pow)</code> . For complex matrices: the determinant is returned as <code>det_pow</code> , <code>det_re</code> , <code>det_im</code> in three consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_re}) + \text{abs}(\text{det_im}) < 10.0$ and <code>determinant = (det_re, det_im) * 10^(det_pow)</code> .	Inertia	Inertia of a real symmetric matrix is defined as a triplet of nonnegative integers (p, n, z) , where p is the number of positive eigenvalues, n is the number of negative eigenvalues, and z is the number of zero eigenvalues. Inertia is returned as three consecutive return array locations p, n, z . Computing inertia can lead to incorrect results for matrixes with a cluster of eigenvalues which are near 0.
ReorderTime	Amount of time taken to do the reordering.										
FactorTime	Amount of time taken to do the factorization.										
SolveTime	Amount of time taken to solve the problem after factorization.										
Determinant	Determinant of the matrix A . For real matrices: the determinant is returned as <code>det_pow</code> , <code>det_base</code> in two consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_base}) < 10.0$ and <code>determinant = det_base * 10^(det_pow)</code> . For complex matrices: the determinant is returned as <code>det_pow</code> , <code>det_re</code> , <code>det_im</code> in three consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_re}) + \text{abs}(\text{det_im}) < 10.0$ and <code>determinant = (det_re, det_im) * 10^(det_pow)</code> .										
Inertia	Inertia of a real symmetric matrix is defined as a triplet of nonnegative integers (p, n, z) , where p is the number of positive eigenvalues, n is the number of negative eigenvalues, and z is the number of zero eigenvalues. Inertia is returned as three consecutive return array locations p, n, z . Computing inertia can lead to incorrect results for matrixes with a cluster of eigenvalues which are near 0.										

Inertia of a k -by- k real symmetric positive definite matrix is always $(k, 0, 0)$. Therefore Inertia is returned only in cases of real symmetric indefinite matrices. For all other matrix types, an error message is returned.

Flops	Number of floating point operations performed during the factorization.
Peakmem	Total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase.
Factormem	Permanent memory in kilobytes that the solver needs from the analysis and symbolic factorization phase in the factorization and solve phases.
Solvemem	Total double precision memory consumption (kilobytes) of the solver for the factorization and solve phases.

NOTE

To avoid problems in passing strings from Fortran to C, Fortran users must call the `mkl_cvt_to_null_terminated_str` routine before calling `dss_statistics`. Refer to the description of `mkl_cvt_to_null_terminated_str` for details.

Output Parameters

`retValues`

FORTRAN 77: REAL*8

Fortran 90: REAL (KIND=8), INTENT (OUT)

Value of the statistics returned.

Finding 'time used to reorder' and 'inertia' of a matrix

The example below illustrates the use of the `dss_statistics` routine.

To find the above values, call `dss_statistics(handle, opt, statArr, retValue)`, where `staArr` is "ReorderTime,Inertia"

In this example, `retValue` has the following values:

<code>retValue[0]</code>	Time to reorder.
<code>retValue[1]</code>	Positive Eigenvalues.
<code>retValue[2]</code>	Negative Eigenvalues.
<code>retValue[3]</code>	Zero Eigenvalues.

Return Values

`MKL_DSS_SUCCESS`

`MKL_DSS_INVALID_OPTION`

`MKL_DSS_STATISTICS_INVALID_MATRIX`

MKL_DSS_STATISTICS_INVALID_STATE
MKL_DSS_STATISTICS_INVALID_STRING
MKL_DSS_MSG_LVL_ERR
MKL_DSS_TERM_LVL_ERR

mkl_cvt_to_null_terminated_str

Passes character strings from Fortran routines to C routines.

Syntax

```
mkl_cvt_to_null_terminated_str (destStr, destLen, srcStr)
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_dss.f90

Description

The routine `mkl_cvt_to_null_terminated_str` passes character strings from Fortran routines to C routines. The strings are converted into integer arrays before being passed to C. Using this routine avoids the problems that can occur on some platforms when passing strings from Fortran to C. The use of this routine is highly recommended.

Input Parameters

<code>destLen</code>	INTEGER. Length of the output array <code>destStr</code> .
<code>srcStr</code>	STRING. Input string.

Output Parameters

<code>destStr</code>	INTEGER. One-dimensional array of integers.
----------------------	---

Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)

Intel MKL supports iterative sparse solvers (ISS) based on the reverse communication interface (RCI), referred to here as the RCI ISS interface. The RCI ISS interface implements a group of user-callable routines that are used in the step-by-step solving process of a symmetric positive definite system (RCI conjugate gradient solver, or RCI CG), and of a non-symmetric indefinite (non-degenerate) system (RCI flexible generalized minimal residual solver, or RCI FGMRES) of linear algebraic equations. This interface uses the general RCI scheme described in [Dong95].

See the [Appendix A Linear Solvers Basics](#) for discussion of terms and concepts related to the ISS routines.

The term *RCI* indicates that when the solver needs the results of certain operations (for example, matrix-vector multiplications), the user performs them and passes the result to the solver. This makes the solver more universal as it is independent of the specific implementation of the operations like the matrix-vector multiplication. To perform such operations, the user can use the built-in sparse matrix-vector multiplications and triangular solvers routines described in [Sparse BLAS Level 2 and Level 3 Routines](#).

NOTE

The RCI CG solver is implemented in two versions: for system of equations with a single right-hand side, and for systems of equations with multiple right-hand sides.

The CG method may fail to compute the solution or compute the wrong solution if the matrix of the system is not symmetric and not positive definite.

The FGMRES method may fail if the matrix is degenerate.

Table "RCI CG Interface Routines" lists the names of the routines, and describes their general use.

RCI ISS Interface Routines

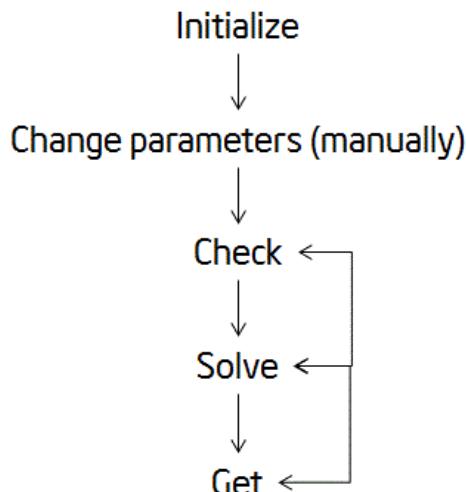
Routine	Description
<code>dcg_init</code> , <code>dcmgrhs_init</code> , <code>dfgmres_init</code>	Initializes the solver.
<code>dcg_check</code> , <code>dcmgrhs_check</code> , <code>dfgmres_check</code>	Checks the consistency and correctness of the user defined data.
<code>dcg</code> , <code>dcmgrhs</code> , <code>dfgmres</code>	Computes the approximate solution vector.
<code>dcg_get</code> , <code>dcmgrhs_get</code> , <code>dfgmres_get</code>	Retrieves the number of the current iteration.

The Intel MKL RCI ISS interface routines are normally invoked in this order:

1. `<system_type>_init`
2. `<system_type>_check`
3. `<system_type>`
4. `<system_type>_get`

Advanced users can change that order if they need it. Others should follow the above order of calls.

The following diagram indicates the typical order in which the RCI ISS interface routines are invoked.

Typical Order for Invoking RCI ISS interface Routines

See the code examples that use the RCI ISS interface routines to solve systems of linear equations in the Intel MKL installation directory.

- Fortran examples: `examples/solverf/source`
- C examples: `examples/solverc/source`

CG Interface Description

All types in this documentation refer to the common Fortran types, INTEGER, and DOUBLE PRECISION.

C and C++ programmers should refer to the section [Calling Sparse Solver and Preconditioner Routines from C/C++](#) for information on mapping Fortran types to C/C++ types.

Each routine for the RCI CG solver is implemented in two versions: for a system of equations with a single right-hand side (SRHS), and for a system of equations with multiple right-hand sides (MRHS). The names of routines for a system with MRHS contain the suffix `mrhs`.

Routine Options

All of the RCI CG routines have common parameters for passing various options to the routines (see [CG Common Parameters](#)). The values for these parameters can be changed during computations.

User Data Arrays

Many of the RCI CG routines take arrays of user data as input. For example, user arrays are passed to the routine `d cg` to compute the solution of a system of linear algebraic equations. The Intel MKL RCI CG routines do not make copies of the user input arrays to minimize storage requirements and improve overall run-time efficiency.

CG Common Parameters

NOTE

The default and initial values listed below are assigned to the parameters by calling the `d cg_init/`
`d cgmrhs_init` routine.

<i>n</i>	INTEGER, this parameter sets the size of the problem in the <code>d cg_init/</code> <code>d cgmrhs_init</code> routine. All the other routines uses the <code>i par(1)</code> parameter instead. Note that the coefficient matrix A is a square matrix of size $n \times n$.
<i>x</i>	DOUBLE PRECISION array of size n for SRHS, or matrix of size $(n \times nrhs)$ for MRHS. This parameter contains the current approximation to the solution. Before the first call to the <code>d cg/d cgmrhs</code> routine, it contains the initial approximation to the solution.
<i>nrhs</i>	INTEGER, this parameter sets the number of right-hand sides for MRHS routines.
<i>b</i>	DOUBLE PRECISION array containing a single right-hand side vector, or matrix of size $n \times nrhs$ containing right-hand side vectors.
<i>RCI_request</i>	INTEGER, this parameter gives information about the result of work of the RCI CG routines. Negative values of the parameter indicate that the routine completed with errors or warnings. The 0 value indicates successful completion of the task. Positive values mean that you must perform specific actions: <code>RCI_request = 1</code> multiply the matrix by $tmp(1:n, 1)$, put the result in $tmp(1:n, 2)$, and return the control to the <code>d cg/</code> <code>d cgmrhs</code> routine;

RCI_request= 2 to perform the stopping tests. If they fail, return the control to the `dchg/dcgmrhs` routine. If the stopping tests succeed, it indicates that the solution is found and stored in the `x` array;

RCI_request= 3 for SRHS: apply the preconditioner to `tmp(1:n, 3)`, put the result in `tmp(1:n, 4)`, and return the control to the `dchg` routine;

for MRHS: apply the preconditioner to `tmp(:, 3+ipar(3))`, put the result in `tmp(:, 3)`, and return the control to the `dcgmrhs` routine.

Note that the `dchg_get/dcgmrhs_get` routine does not change the parameter *RCI_request*. This enables use of this routine inside the *reverse communication* computations.

ipar INTEGER array, of size 128 for SRHS, and of size $(128+2*nrhs)$ for MRHS. This parameter specifies the integer set of data for the RCI CG computations:

ipar(1) specifies the size of the problem. The `dchg_init/dcgmrhs_init` routine assigns *ipar(1)=n*. All the other routines use this parameter instead of *n*. There is no default value for this parameter.

ipar(2) specifies the type of output for error and warning messages generated by the RCI CG routines. The default value 6 means that all messages are displayed on the screen. Otherwise, the error and warning messages are written to the newly created files `dchg_errors.txt` and `dchg_check_warnings.txt`, respectively. Note that if *ipar(6)* and *ipar(7)* parameters are set to 0, error and warning messages are not generated at all.

ipar(3) for SRHS: contains the current stage of the RCI CG computations. The initial value is 1;
for MRHS: contains the number of the right-hand side for which the calculations are currently performed.

WARNING

Avoid altering this variable during computations.

ipar(4) contains the current iteration number. The initial value is 0.

ipar(5) specifies the maximum number of iterations. The default value is `min(150, n)`.

ipar(6)

if the value is not equal to 0, the routines output error messages in accordance with the parameter *ipar(2)*. Otherwise, the routines do not output error messages at all, but return a negative value of the parameter *RCI_request*. The default value is 1.

ipar(7)

if the value is not equal to 0, the routines output warning messages in accordance with the parameter *ipar(2)*. Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter *RCI_request*. The default value is 1.

ipar(8)

if the value is not equal to 0, the *d_cg/d_cgmrhs* routine performs the stopping test for the maximum number of iterations: $ipar(4) \leq ipar(5)$. Otherwise, the method is stopped and the corresponding value is assigned to the *RCI_request*. If the value is 0, the routine does not perform this stopping test. The default value is 1.

ipar(9)

if the value is not equal to 0, the *d_cg/d_cgmrhs* routine performs the residual stopping test:
 $dpar(5) \leq dpar(4) = dpar(1) * dpar(3) + dpar(2)$. Otherwise, the method is stopped and corresponding value is assigned to the *RCI_request*. If the value is 0, the routine does not perform this stopping test. The default value is 0.

ipar(10)

if the value is not equal to 0, the *d_cg/d_cgmrhs* routine requests a user-defined stopping test by setting the output parameter *RCI_request*=2. If the value is 0, the routine does not perform the user defined stopping test. The default value is 1.

NOTE

At least one of the parameters *ipar(8)*-
ipar(10) must be set to 1.

ipar(11)

if the value is equal to 0, the *d_cg/d_cgmrhs* routine runs the non-preconditioned version of the corresponding CG method. Otherwise, the routine runs the preconditioned version of the CG method, and by setting the output parameter *RCI_request*=3, indicates that you must perform the preconditioning step. The default value is 0.

ipar(11:128)

are reserved and not used in the current RCI CG SRHS and MRHS routines.

NOTE

For future compatibility, you must declare the array *ipar* with length 128 for a single right-hand side.

ipar
(11:128+2**nrhs*)

are reserved for internal use in the current RCI CG SRHS and MRHS routines.

NOTE

For future compatibility, you must declare the array *ipar* with length 128+2**nrhs* for multiple right-hand sides.

dpar

DOUBLE PRECISION array, for SRHS of size 128, for MRHS of size (128+2**nrhs*) ; this parameter is used to specify the double precision set of data for the RCI CG computations, specifically:

<i>dpar</i> (1)	specifies the relative tolerance. The default value is 1.0X10 ⁻⁶ .
<i>dpar</i> (2)	specifies the absolute tolerance. The default value is 0.0.
<i>dpar</i> (3)	specifies the square norm of the initial residual (if it is computed in the <i>drg/dcgmrhs</i> routine). The initial value is 0.0.
<i>dpar</i> (4)	service variable equal to <i>dpar</i> (1) * <i>dpar</i> (3) + <i>dpar</i> (2) (if it is computed in the <i>drg/dcgmrhs</i> routine). The initial value is 0.0.
<i>dpar</i> (5)	specifies the square norm of the current residual. The initial value is 0.0.
<i>dpar</i> (6)	specifies the square norm of residual from the previous iteration step (if available). The initial value is 0.0.
<i>dpar</i> (7)	contains the <i>alpha</i> parameter of the CG method. The initial value is 0.0.
<i>dpar</i> (8)	contains the <i>beta</i> parameter of the CG method, it is equal to <i>dpar</i> (5) / <i>dpar</i> (6) The initial value is 0.0.
<i>dpar</i> (9:128)	are reserved and not used in the current RCI CG SRHS and MRHS routines.

NOTE

For future compatibility, you must declare the array *dpar* with length 128 for a single right-hand side.

dpar(9:128+2**nrhs*) are reserved for internal use in the current RCI CG SRHS and MRHS routines.

NOTE

For future compatibility, you must declare the array *dpar* with length 128+2**nrhs* for multiple right-hand sides.

tmp

DOUBLE PRECISION array of size (*n*, 4) for SRHS, and (*n*, 3+*nrhs*) for MRHS. This parameter is used to supply the double precision temporary space for the RCI CG computations, specifically:

<i>tmp</i> (:, 1)	specifies the current search direction. The initial value is 0.0.
<i>tmp</i> (:, 2)	contains the matrix multiplied by the current search direction. The initial value is 0.0.
<i>tmp</i> (:, 3)	contains the current residual. The initial value is 0.0.
<i>tmp</i> (:, 4)	contains the inverse of the preconditioner applied to the current residual for the SRHS version of CG. There is no initial value for this parameter.
<i>tmp</i> (:, 4:3+ <i>nrhs</i>)	contains the inverse of the preconditioner applied to the current residual for the MRHS version of CG. There is no initial value for this parameter.

NOTE

You can define this array in the code using RCI CG SRHS as DOUBLE PRECISION *tmp*(*n*, 3) if you run only non-preconditioned CG iterations.

Schemes of Using the RCI CG Routines

The following pseudocode shows the general schemes of using the RCI CG routines for the SRHS case. The MRHS is similar (see the example code for more details).

```
...
generate matrix A
generate preconditioner C (optional)
call dcg_init(n, x, b, RCI_request, ipar, dpar, tmp)
change parameters in ipar, dpar if necessary
call dcg_check(n, x, b, RCI_request, ipar, dpar, tmp)
```

```

1 call dcg(n, x, b, RCI_request, ipar, dpar, tmp)
  if (RCI_request.eq.1) then
    multiply the matrix A by tmp(1:n,1) and put the result in tmp(1:n,2)
    It is possible to use MKL Sparse BLAS Level 2 subroutines for this operation
  c proceed with CG iterations
    goto 1
  endif
  if (RCI_request.eq.2) then
    do the stopping test
    if (test not passed) then
  c proceed with CG iterations
    go to 1
  else
  c stop CG iterations
    goto 2
  endif
  endif
  if (RCI_request.eq.3) then (optional)
    apply the preconditioner C inverse to tmp(1:n,3) and put the result in tmp(1:n,4)
  c proceed with CG iterations
    goto 1
  end
2 call dcg_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)
  current iteration number is in itercount
  the computed approximation is in the array x

```

FGMRES Interface Description

All types in this documentation refer to the common Fortran types: INTEGER and DOUBLE PRECISION.

C and C++ programmers should refer to the section [Calling Sparse Solver and Preconditioner Routines from C/C++](#) for information on mapping Fortran types to C/C++ types.

Routine Options

All of the RCI FGMRES routines have common parameters for passing various options to the routines (see [FGMRES Common Parameters](#)). The values for these parameters can be changed during computations.

User Data Arrays

Many of the RCI FGMRES routines take arrays of user data as input. For example, user arrays are passed to the routine `dfgmres` to compute the solution of a system of linear algebraic equations. To minimize storage requirements and improve overall run-time efficiency, the Intel MKL RCI FGMRES routines do not make copies of the user input arrays.

FGMRES Common Parameters

NOTE

The default and initial values listed below are assigned to the parameters by calling the `dfgmres_init` routine.

<i>n</i>	INTEGER, this parameter sets the size of the problem in the <code>dfgmres_init</code> routine. All the other routines uses <code>ipar(1)</code> parameter instead. Note that the coefficient matrix A is a square matrix of size $n \times n$.								
<i>x</i>	DOUBLE PRECISION array, this parameter contains the current approximation to the solution vector. Before the first call to the <code>dfgmres</code> routine, it contains the initial approximation to the solution vector.								
<i>b</i>	DOUBLE PRECISION array, this parameter contains the right-hand side vector. Depending on user requests (see the parameter <code>ipar(13)</code>), it might contain the approximate solution after execution.								
<i>RCI_request</i>	INTEGER, this parameter gives information about the result of work of the RCI FGMRES routines. Negative values of the parameter indicate that the routine completed with errors or warnings. The 0 value indicates successful completion of the task. Positive values mean that you must perform specific actions: <table border="0"> <tbody> <tr> <td><i>RCI_request</i>= 1</td><td>multiply the matrix by <code>tmp(ipar(22))</code>, put the result in <code>tmp(ipar(23))</code>, and return the control to the <code>dfgmres</code> routine;</td></tr> <tr> <td><i>RCI_request</i>= 2</td><td>perform the stopping tests. If they fail, return the control to the <code>dfgues</code> routine. Otherwise, the solution can be updated by a subsequent call to <code>dfgmres_get</code> routine;</td></tr> <tr> <td><i>RCI_request</i>= 3</td><td>apply the preconditioner to <code>tmp(ipar(22))</code>, put the result in <code>tmp(ipar(23))</code>, and return the control to the <code>dfgmres</code> routine.</td></tr> <tr> <td><i>RCI_request</i>= 4</td><td>check if the norm of the current orthogonal vector is zero, within the rounding or computational errors. Return the control to the <code>dfgmres</code> routine if it is not zero, otherwise complete the solution process by calling <code>dfgmres_get</code> routine.</td></tr> </tbody> </table>	<i>RCI_request</i> = 1	multiply the matrix by <code>tmp(ipar(22))</code> , put the result in <code>tmp(ipar(23))</code> , and return the control to the <code>dfgmres</code> routine;	<i>RCI_request</i> = 2	perform the stopping tests. If they fail, return the control to the <code>dfgues</code> routine. Otherwise, the solution can be updated by a subsequent call to <code>dfgmres_get</code> routine;	<i>RCI_request</i> = 3	apply the preconditioner to <code>tmp(ipar(22))</code> , put the result in <code>tmp(ipar(23))</code> , and return the control to the <code>dfgmres</code> routine.	<i>RCI_request</i> = 4	check if the norm of the current orthogonal vector is zero, within the rounding or computational errors. Return the control to the <code>dfgmres</code> routine if it is not zero, otherwise complete the solution process by calling <code>dfgmres_get</code> routine.
<i>RCI_request</i> = 1	multiply the matrix by <code>tmp(ipar(22))</code> , put the result in <code>tmp(ipar(23))</code> , and return the control to the <code>dfgmres</code> routine;								
<i>RCI_request</i> = 2	perform the stopping tests. If they fail, return the control to the <code>dfgues</code> routine. Otherwise, the solution can be updated by a subsequent call to <code>dfgmres_get</code> routine;								
<i>RCI_request</i> = 3	apply the preconditioner to <code>tmp(ipar(22))</code> , put the result in <code>tmp(ipar(23))</code> , and return the control to the <code>dfgmres</code> routine.								
<i>RCI_request</i> = 4	check if the norm of the current orthogonal vector is zero, within the rounding or computational errors. Return the control to the <code>dfgmres</code> routine if it is not zero, otherwise complete the solution process by calling <code>dfgmres_get</code> routine.								
<i>ipar(128)</i>	INTEGER array, this parameter specifies the integer set of data for the RCI FGMRES computations:								
<i>ipar(1)</i>	specifies the size of the problem. The <code>dfgmres_init</code> routine assigns <code>ipar(1)=n</code> . All the other routines uses this parameter instead of <i>n</i> . There is no default value for this parameter.								

ipar(2)

specifies the type of output for error and warning messages that are generated by the RCI FGMRES routines. The default value 6 means that all messages are displayed on the screen. Otherwise the error and warning messages are written to the newly created file `MKL_RCI_FGMRES_Log.txt`. Note that if *ipar(6)* and *ipar(7)* parameters are set to 0, error and warning messages are not generated at all.

ipar(3)

contains the current stage of the RCI FGMRES computations., The initial value is 1.

WARNING

Avoid altering this variable during computations.

ipar(4)

contains the current iteration number. The initial value is 0.

ipar(5)

specifies the maximum number of iterations. The default value is *min* (150,*n*).

ipar(6)

if the value is not 0, the routines output error messages in accordance with the parameter *ipar(2)*. If it is 0, the routines do not output error messages at all, but return a negative value of the parameter `RCI_request`. The default value is 1.

ipar(7)

if the value is not 0, the routines output warning messages in accordance with the parameter *ipar(2)*. Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter `RCI_request`. The default value is 1.

ipar(8)

if the value is not equal to 0, the `dfgmres` routine performs the stopping test for the maximum number of iterations: $ipar(4) \leq ipar(5)$. If the value is 0, the `dfgmres` routine does not perform this stopping test. The default value is 1.

ipar(9)

if the value is not 0, the `dfgmres` routine performs the residual stopping test: $dpar(5) \leq dpar(4)$.If the value is 0, the `dfgmres` routine does not perform this stopping test. The default value is 0.

ipar(10)

if the value is not 0, the `dfgmres` routine indicates that the user-defined stopping test should be performed by setting `RCI_request=2`. If the value is 0, the `dfgmres` routine does not perform the user-defined stopping test. The default value is 1.

NOTE

At least one of the parameters $ipar(8)$ - $ipar(10)$ must be set to 1.

$ipar(11)$

if the value is 0, the `dfgmres` routine runs the non-preconditioned version of the FGMRES method. Otherwise, the routine runs the preconditioned version of the FGMRES method, and requests that you perform the preconditioning step by setting the output parameter `RCI_request=3`. The default value is 0.

$ipar(12)$

if the value is not equal to 0, the `dfgmres` routine performs the automatic test for zero norm of the currently generated vector: $dpar(7) \leq dpar(8)$, where $dpar(8)$ contains the tolerance value. Otherwise, the routine indicates that you must perform this check by setting the output parameter `RCI_request=4`. The default value is 0.

$ipar(13)$

if the value is equal to 0, the `dfgmres_get` routine updates the solution to the vector x according to the computations done by the `dfgmres` routine. If the value is positive, the routine writes the solution to the right-hand side vector b . If the value is negative, the routine returns only the number of the current iteration, and does not update the solution. The default value is 0.

NOTE

It is possible to call the `dfgmres_get` routine at any place in the code, but you must pay special attention to the parameter $ipar(13)$. The RCI FGMRES iterations can be continued after the call to `dfgmres_get` routine only if the parameter $ipar(13)$ is not equal to zero. If $ipar(13)$ is positive, then the updated solution overwrites the right-hand side in the vector b . If you want to run the restarted version of FGMRES with the same right-hand side, then it must be saved in a different memory location before the first call to the `dfgmres_get` routine with positive $ipar(13)$.

$ipar(14)$

contains the internal iteration counter that counts the number of iterations before the restart takes place. The initial value is 0.

WARNING

Do not alter this variable during computations.

ipar(15)

specifies the number of the non-restarted FGMRES iterations. To run the restarted version of the FGMRES method, assign the number of iterations to *ipar*(15) before the restart. The default value is $\min(150, n)$, which means that by default the non-restarted version of FGMRES method is used.

ipar(16)

service variable specifying the location of the rotated Hessenberg matrix from which the matrix stored in the packed format (see [Matrix Arguments](#) in the Appendix B for details) is started in the *tmp* array.

ipar(17)

service variable specifying the location of the rotation cosines from which the vector of cosines is started in the *tmp* array.

ipar(18)

service variable specifying the location of the rotation sines from which the vector of sines is started in the *tmp* array.

ipar(19)

service variable specifying the location of the rotated residual vector from which the vector is started in the *tmp* array.

ipar(20)

service variable, specifies the location of the least squares solution vector from which the vector is started in the *tmp* array.

ipar(21)

service variable specifying the location of the set of preconditioned vectors from which the set is started in the *tmp* array. The memory locations in the *tmp* array starting from *ipar*(21) are used only for the preconditioned FGMRES method.

ipar(22)

specifies the memory location from which the first vector (source) used in operations requested via *RCI_request* is started in the *tmp* array.

ipar(23)

specifies the memory location from which the second vector (output) used in operations requested via *RCI_request* is started in the *tmp* array.

ipar(24:128)

are reserved and not used in the current RCI FGMRES routines.

NOTE

You must declare the array *ipar* with length 128. While defining the array in the code as INTEGER *ipar*(23) works, there is no guarantee of future compatibility with Intel MKL.

dpar(128)

DOUBLE PRECISION array, this parameter specifies the double precision set of data for the RCI CG computations, specifically:

dpar(1)

specifies the relative tolerance. The default value is 1.0D-6.

dpar(2)

specifies the absolute tolerance. The default value is 0.0D-0.

dpar(3)

specifies the Euclidean norm of the initial residual (if it is computed in the *dfgmres* routine). The initial value is 0.0.

dpar(4)

service variable equal to *dpar*(1) **dpar*(3) +*dpar*(2) (if it is computed in the *dfgmres* routine). The initial value is 0.0.

dpar(5)

specifies the Euclidean norm of the current residual. The initial value is 0.0.

dpar(6)

specifies the Euclidean norm of residual from the previous iteration step (if available). The initial value is 0.0.

dpar(7)

contains the norm of the generated vector. The initial value is 0.0.

dpar(8)

contains the tolerance for the zero norm of the currently generated vector. The default value is 1.0D-12.

dpar(9:128)

are reserved and not used in the current RCI FGMRES routines.

NOTE

You must declare the array *dpar* with length 128. While defining the array in the code as DOUBLE PRECISION *dpar*(8) works, there is no guarantee of future compatibility with Intel MKL.

<i>tmp</i>	DOUBLE PRECISION array of size $((2*ipar(15)+1)*n + ipar(15)*(ipar(15)+9)/2 + 1)$ used to supply the double precision temporary space for the RCI FGMRES computations, specifically:
<i>tmp</i> $(1:ipar(16)-1)$	contains the sequence of vectors generated by the FGMRES method. The initial value is 0.0.
<i>tmp</i> $(ipar(16):ipar(17))$	contains the rotated Hessenberg matrix generated by the FGMRES method; the matrix is stored in the packed format. There is no initial value for this part of <i>tmp</i> array.
<i>tmp</i> $(ipar(17):ipar(18))$	contains the rotation cosines vector generated by the FGMRES method. There is no initial value for this part of <i>tmp</i> array.
<i>tmp</i> $(ipar(18):ipar(19))$	contains the rotation sines vector generated by the FGMRES method. There is no initial value for this part of <i>tmp</i> array.
<i>tmp</i> $(ipar(19):ipar(20))$	contains the rotated residual vector generated by the FGMRES method. There is no initial value for this part of <i>tmp</i> array.
<i>tmp</i> $(ipar(20):ipar(21))$	contains the solution vector to the least squares problem generated by the FGMRES method. There is no initial value for this part of <i>tmp</i> array.
<i>tmp</i> $(ipar(21):)$	contains the set of preconditioned vectors generated for the FGMRES method by the user. This part of <i>tmp</i> array is not used if the non-preconditioned version of FGMRES method is called. There is no initial value for this part of <i>tmp</i> array.

NOTE

You can define this array in the code as
 DOUBLE PRECISION $tmp((2*ipar(15)+1)*n + ipar(15)*(ipar(15)+9)/2 + 1)$ if you run only non-preconditioned FGMRES iterations.

Scheme of Using the RCI FGMRES Routines

The following pseudocode shows the general scheme of using the RCI FGMRES routines.

```

...
generate matrix A
generate preconditioner C (optional)
call dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
  change parameters in ipar, dpar if necessary
call dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)
1 call dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
  if (RCI_request.eq.1) then

```

multiply the matrix A by $\text{tmp}(ipar(22))$ and put the result in $\text{tmp}(ipar(23))$

It is possible to use [MKL Sparse BLAS Level 2](#) subroutines for this operation

c proceed with FGMRES iterations

 goto 1

 endif

 if (RCI_request.eq.2) then

 do the stopping test

 if (test not passed) then

c proceed with FGMRES iterations

 go to 1

 else

c stop FGMRES iterations

 goto 2

 endif

 endif

 if (RCI_request.eq.3) then (optional)

 apply the preconditioner C inverse to $\text{tmp}(ipar(22))$ and put the result in $\text{tmp}(ipar(23))$

c proceed with FGMRES iterations

 goto 1

 endif

 if (RCI_request.eq.4) then

 check the norm of the next orthogonal vector, it is contained in $dpar(7)$

 if (the norm is not zero up to rounding/computational errors) then

c proceed with FGMRES iterations

 goto 1

 else

c stop FGMRES iterations

 goto 2

 endif

 endif

2 call dfgmres_get(n , x , b , RCI_request, ipar, dpar, tmp, itercount)

current iteration number is in $itercount$

the computed approximation is in the array x

NOTE

For the FGMRES method, the array *x* initially contains the current approximation to the solution. It can be updated only by calling the routine `dfgmres_get`, which updates the solution in accordance with the computations performed by the routine `dfgmres`.

The above pseudocode demonstrates two main differences in the use of RCI FGMRES interface comparing with the [CG Interface Description](#). The first difference relates to *RCI_request*=3: it uses different locations in the *tmp* array, which is two-dimensional for CG and one-dimensional for FGMRES. The second difference relates to *RCI_request*=4: the RCI CG interface never produces *RCI_request*=4.

RCI ISS Routines

`dcg_init`

Initializes the solver.

Syntax

Fortran:

```
dcg_init(n, x, b, RCI_request, ipar, dpar, tmp)
```

C:

```
void dcg_init (MKL_INT *n, double *x, double *b, MKL_INT *RCI_request, MKL_INT *ipar,
double *dpar, double *tmp);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The routine `dcg_init` initializes the solver. After initialization, all subsequent invocations of the Intel MKL RCI CG routines use the values of all parameters returned by the routine `dcg_init`. Advanced users can skip this step and set the values in the *ipar* and *dpar* arrays directly.

CAUTION

You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dcg_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

n INTEGER. Sets the size of the problem.

x DOUBLE PRECISION. Array of size *n*. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to *b*.

b DOUBLE PRECISION. Array of size *n*. Contains the right-hand side vector.

Output Parameters

<i>RCI_request</i>	INTEGER. Gives information about the result of the routine.
<i>ipar</i>	INTEGER . Array of size 128. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size 128. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION . Array of size $(n, 4)$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -10000	Indicates failure to complete the task.

dcg_check

Checks consistency and correctness of the user defined data.

Syntax

Fortran:

```
dcg_check(n, x, b, RCI_request, ipar, dpar, tmp)
```

C:

```
void dcg_check (MKL_INT *n, double *x, double *b, MKL_INT *RCI_request, MKL_INT *ipar,  
double *dpar, double *tmp);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The routine `dcg_check` checks consistency and correctness of the parameters to be passed to the solver routine `dcg`. However this operation does not guarantee that the solver returns the correct result. It only reduces the chance of making a mistake in the parameters of the method. Skip this operation only if you are sure that the correct data is specified in the solver parameters.

The lengths of all vectors must be defined in a previous call to the `dcg_init` routine.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>x</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to <i>b</i> .
<i>b</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the right-hand side vector.

Output Parameters

<i>RCI_request</i>	INTEGER. Gives information about result of the routine.
<i>ipar</i>	INTEGER. Array of size 128. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size 128. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size (<i>n</i> , 4). Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -1100	Indicates that the task is interrupted and the errors occur.
<i>RCI_request</i> = -1001	Indicates that there are some warning messages.
<i>RCI_request</i> = -1010	Indicates that the routine changed some parameters to make them consistent or correct.
<i>RCI_request</i> = -1011	Indicates that there are some warning messages and that the routine changed some parameters.

dcg

Computes the approximate solution vector.

Syntax

Fortran:

```
dcg(n, x, b, RCI_request, ipar, dpar, tmp)
```

C:

```
void dcg (MKL_INT *n, double *x, double *b, MKL_INT *RCI_request, MKL_INT *ipar,
double *dpar, double *tmp);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The `dcg` routine computes the approximate solution vector using the CG method [[Young71](#)]. The routine `dcg` uses the vector in the array *x* before the first call as an initial approximation to the solution. The parameter *RCI_request* gives you information about the task completion and requests results of certain operations that are required by the solver.

Note that lengths of all vectors must be defined in a previous call to the `dcg_init` routine.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
----------	--

<i>x</i>	DOUBLE PRECISION . Array of size <i>n</i> . Contains the initial approximation to the solution vector.
<i>b</i>	DOUBLE PRECISION . Array of size <i>n</i> . Contains the right-hand side vector.
<i>tmp</i>	DOUBLE PRECISION . Array of size (<i>n</i> , 4) . Refer to the CG Common Parameters .

Output Parameters

<i>RCI_request</i>	INTEGER. Gives information about result of work of the routine.
<i>x</i>	DOUBLE PRECISION . Array of size <i>n</i> . Contains the updated approximation to the solution vector.
<i>ipar</i>	INTEGER . Array of size 128. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION . Array of size 128. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION . Array of size (<i>n</i> , 4) . Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> =0	Indicates that the task completed normally and the solution is found and stored in the vector <i>x</i> . This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the <i>RCI_request</i> = 2.
<i>RCI_request</i> =-1	Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if you request both tests.
<i>RCI_request</i> =-2	Indicates that the routine was interrupted because of an attempt to divide by zero. This situation happens if the matrix is non-positive definite or almost non-positive definite.
<i>RCI_request</i> =- 10	Indicates that the routine was interrupted because the residual norm is invalid. This usually happens because the value <i>dpar</i> (6) was altered outside of the routine, or the <i>drg_check</i> routine was not called.
<i>RCI_request</i> =-11	Indicates that the routine was interrupted because it enters the infinite cycle. This usually happens because the values <i>ipar</i> (8), <i>ipar</i> (9), <i>ipar</i> (10) were altered outside of the routine, or the <i>drg_check</i> routine was not called.
<i>RCI_request</i> = 1	Indicates that you must multiply the matrix by <i>tmp</i> (1: <i>n</i> , 1), put the result in the <i>tmp</i> (1: <i>n</i> , 2), and return the control back to the routine <i>drg</i> .

<i>RCI_request</i> = 2	Indicates that you must perform the stopping tests. If they fail, return control back to the <code>dcg</code> routine. Otherwise, the solution is found and stored in the vector <i>x</i> .
<i>RCI_request</i> = 3	Indicates that you must apply the preconditioner to <i>tmp</i> (:, 3), put the result in the <i>tmp</i> (:, 4), and return the control back to the routine <code>dcg</code> .

dcg_get

Retrieves the number of the current iteration.

Syntax

Fortran:

```
dcg_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)
```

C:

```
void dcg_get (MKL_INT *n, double *x, double *b, MKL_INT *RCI_request, MKL_INT *ipar,  
double *dpar, double *tmp, MKL_INT *itercount);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The routine `dcg_get` retrieves the current iteration number of the solutions process.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>x</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the approximation vector to the solution.
<i>b</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the right-hand side vector.
<i>RCI_request</i>	INTEGER. This parameter is not used.
<i>ipar</i>	INTEGER. Array of size 128. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size 128. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size (<i>n</i> , 4). Refer to the CG Common Parameters .

Output Parameters

<i>itercount</i>	INTEGER. Returns the current iteration number.
------------------	--

Return Values

The routine `dcg_get` has no return values.

dchgmrhs_init

Initializes the RCI CG solver with MRHS.

Syntax

Fortran:

```
dchgmrhs_init(n, x, nrhs, b, method, RCI_request, ipar, dpar, tmp)
```

C:

```
void dchgmrhs_init (MKL_INT *n, double *x, MKL_INT *nrhs, double *b, MKL_INT *method,  
MKL_INT *RCI_request, MKL_INT *ipar, double *dpar, double *tmp);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The routine `dchgmrhs_init` initializes the solver. After initialization all subsequent invocations of the Intel MKL RCI CG with multiple right-hand sides (MRHS) routines use the values of all parameters that are returned by `dchgmrhs_init`. Advanced users may skip this step and set the values to these parameters directly in the appropriate routines.

WARNING

You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dchgmrhs_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>x</i>	DOUBLE PRECISION. Array of size $n \times nrhs$. Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to <i>b</i> .
<i>nrhs</i>	INTEGER. Sets the number of right-hand sides.
<i>b</i>	DOUBLE PRECISION. Array of size $n \times nrhs$. Contains the right-hand side vectors.
<i>method</i>	INTEGER. Specifies the method of solution: A value of 1 indicates CG with multiple right-hand sides (default value)

Output Parameters

<i>RCI_request</i>	INTEGER. Gives information about the result of the routine.
<i>ipar</i>	INTEGER. Array of size $(128 + 2 \times nrhs)$. Refer to the CG Common Parameters .

<i>dpar</i>	DOUBLE PRECISION. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $(n, 3+nrhs)$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -10000	Indicates failure to complete the task.

dcgmrhs_check

Checks consistency and correctness of the user defined data.

Syntax

Fortran:

```
dcgmrhs_check(n, x, nrhs, b, RCI_request, ipar, dpar, tmp)
```

C:

```
void dcgmrhs_check (MKL_INT *n, double *x, MKL_INT *nrhs, double *b, MKL_INT  
*RCI_request, MKL_INT *ipar, double *dpar, double *tmp);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The routine `dcgmrhs_check` checks the consistency and correctness of the parameters to be passed to the solver routine `dcgmrhs`. While this operation reduces the chance of making a mistake in the parameters, it does not guarantee that the solver returns the correct result.

If you are sure that the correct data is *specified* in the solver parameters, you can skip this operation.

The lengths of all vectors must be defined in a previous call to the `dcgmrhs_init` routine.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>x</i>	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to <i>b</i> .
<i>nrhs</i>	INTEGER. This parameter sets the number of right-hand sides.
<i>b</i>	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the right-hand side vectors.

Output Parameters

<i>RCI_request</i>	INTEGER. Returns information about the results of the routine.
<i>ipar</i>	INTEGER. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $(n, 3+nrhs)$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -1100	Indicates that the task is interrupted and the errors occur.
<i>RCI_request</i> = -1001	Indicates that there are some warning messages.
<i>RCI_request</i> = -1010	Indicates that the routine changed some parameters to make them consistent or correct.
<i>RCI_request</i> = -1011	Indicates that there are some warning messages and that the routine changed some parameters.

dgmrhs

Computes the approximate solution vectors.

Syntax

Fortran:

```
dgmrhs(n, x, nrhs, b, RCI_request, ipar, dpar, tmp)
```

C:

```
void dgmrhs (MKL_INT *n, double *x, MKL_INT *nrhs, double *b, MKL_INT *RCI_request,
MKL_INT *ipar, double *dpar, double *tmp);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The routine `dgmrhs` computes approximate solution vectors using the CG with multiple right-hand sides (MRHS) method [Young71]. The routine `dgmrhs` uses the value that was in the `x` before the first call as an initial approximation to the solution. The parameter `RCI_request` gives information about task completion status and requests results of certain operations that are required by the solver.

Note that lengths of all vectors are assumed to have been defined in a previous call to the `dgmrhs_init` routine.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem, and the sizes of arrays <i>x</i> and <i>b</i> .
<i>x</i>	DOUBLE PRECISION. Array of size $n \times nrhs$. Contains the initial approximation to the solution vectors.
<i>nrhs</i>	INTEGER. Sets the number of right-hand sides.
<i>b</i>	DOUBLE PRECISION. Array of size $n \times nrhs$. Contains the right-hand side vectors.
<i>tmp</i>	DOUBLE PRECISION. Array of size $(n, 3+nrhs)$. Refer to the CG Common Parameters .

Output Parameters

<i>RCI_request</i>	INTEGER. Gives information about result of work of the routine.
<i>x</i>	DOUBLE PRECISION. Array of size n -by- $nrhs$. Contains the updated approximation to the solution vectors.
<i>ipar</i>	INTEGER. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $(n, 3+nrhs)$. Refer to the CG Common Parameters .

Return Values

<i>RCI_request=0</i>	Indicates that the task completed normally and the solution is found and stored in the vector <i>x</i> . This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the <i>RCI_request= 2</i> .
<i>RCI_request=-1</i>	Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if both tests are requested by the user.
<i>RCI_request=-2</i>	The routine was interrupted because of an attempt to divide by zero. This situation happens if the matrix is non-positive definite or almost non-positive definite.
<i>RCI_request=-10</i>	Indicates that the routine was interrupted because the residual norm is invalid. This usually happens because the value <i>dpar(6)</i> was altered outside of the routine, or the <i>dgc_check</i> routine was not called.

RCI_request=-11

Indicates that the routine was interrupted because it enters the infinite cycle. This usually happens because the values $ipar(8)$, $ipar(9)$, $ipar(10)$ were altered outside of the routine, or the `drg_check` routine was not called.

RCI_request= 1

Indicates that you must multiply the matrix by $tmp(1:n, 1)$, put the result in the $tmp(1:n, 2)$, and return the control back to the routine `drg`.

RCI_request= 2

Indicates that you must perform the stopping tests. If they fail, return control back to the `drg` routine. Otherwise, the solution is found and stored in the vector x .

RCI_request= 3

Indicates that you must apply the preconditioner to $tmp(:, 3)$, put the result in the $tmp(:, 4)$, and return the control back to the routine `drg`.

drgmrhs_get

Retrieves the number of the current iteration.

Syntax

Fortran:

```
drgmrhs_get(n, x, nrhs, b, RCI_request, ipar, dpar, tmp, itercount)
```

C:

```
void drgmrhs_get (MKL_INT *n, double *x, MKL_INT *nrhs, double *b, MKL_INT
*RCI_request, MKL_INT *ipar, double *dpar, double *tmp, MKL_INT *itercount);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The routine `drgmrhs_get` retrieves the current iteration number of the solving process.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>x</i>	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the initial approximation to the solution vectors.
<i>nrhs</i>	INTEGER. Sets the number of right-hand sides.
<i>b</i>	DOUBLE PRECISION. Array of size $n*nrhs$. Contains the right-hand side .
<i>RCI_request</i>	INTEGER. This parameter is not used.
<i>ipar</i>	INTEGER. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .

<i>dpar</i>	DOUBLE PRECISION. Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $(n, 3+nrhs)$. Refer to the CG Common Parameters .

Output Parameters

<i>itercount</i>	INTEGER. Array of size $nrhs$. Returns the current iteration number for each right-hand side.
------------------	--

Return Values

The routine `dgmres_get` has no return values.

dfgmres_init

Initializes the solver.

Syntax

Fortran:

```
dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
```

C:

```
void dfgmres_init (MKL_INT *n, double *x, double *b, MKL_INT *RCI_request, MKL_INT  
*ipar, double *dpar, double *tmp);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The routine `dfgmres_init` initializes the solver. After initialization all subsequent invocations of Intel MKL RCI FGMRES routines use the values of all parameters that are returned by `dfgmres_init`. Advanced users can skip this step and set the values in the `ipar` and `dpar` arrays directly.

WARNING

You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dfgmres_check` routine, but it does not guarantee that the method will work correctly.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>x</i>	DOUBLE PRECISION. Array of size n . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to b .
<i>b</i>	DOUBLE PRECISION. Array of size n . Contains the right-hand side vector.

Output Parameters

<i>RCI_request</i>	INTEGER. Gives information about the result of the routine.
<i>ipar</i>	INTEGER. Array of size 128. Refer to the FGMRES Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size 128. Refer to the FGMRES Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $((2*ipar(15) + 1)*n + ipar(15)*(ipar(15) + 9)/2 + 1)$. Refer to the FGMRES Common Parameters .

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -10000	Indicates failure to complete the task.

dfgmres_check

Checks consistency and correctness of the user defined data.

Syntax

Fortran:

```
dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)
```

C:

```
void dfgmres_check (MKL_INT *n, double *x, double *b, MKL_INT *RCI_request, MKL_INT *ipar, double *dpar, double *tmp);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The routine `dfgmres_check` checks consistency and correctness of the parameters to be passed to the solver routine `dfgmres`. However, this operation does not guarantee that the method gives the correct result. It only reduces the chance of making a mistake in the parameters of the routine. Skip this operation only if you are sure that the correct data is specified in the solver parameters.

The lengths of all vectors are assumed to have been defined in a previous call to the `dfgmres_init` routine.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>x</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the initial approximation to the solution vector. Normally it is equal to 0 or to <i>b</i> .
<i>b</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the right-hand side vector.

Output Parameters

<i>RCI_request</i>	INTEGER. Gives information about result of the routine.
<i>ipar</i>	INTEGER. Array of size 128. Refer to the FGMRES Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size 128. Refer to the FGMRES Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $((2*ipar(15)+1)*n + ipar(15)*ipar(15)+9)/2 + 1$. Refer to the FGMRES Common Parameters .

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -1100	Indicates that the task is interrupted and the errors occur.
<i>RCI_request</i> = -1001	Indicates that there are some warning messages.
<i>RCI_request</i> = -1010	Indicates that the routine changed some parameters to make them consistent or correct.
<i>RCI_request</i> = -1011	Indicates that there are some warning messages and that the routine changed some parameters.

dfgmres

Makes the FGMRES iterations.

Syntax

Fortran:

```
dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
```

C:

```
void dfgmres (MKL_INT *n, double *x, double *b, MKL_INT *RCI_request, MKL_INT *ipar,
double *dpar, double *tmp);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The routine `dfgmres` performs the FGMRES iterations [Saad03], using the value that was in the array *x* before the first call as an initial approximation of the solution vector. To update the current approximation to the solution, the `dfgmres_get` routine must be called. The RCI FGMRES iterations can be continued after the call to the `dfgmres_get` routine only if the value of the parameter *ipar*(13) is not equal to 0 (default value). Note that the updated solution overwrites the right-hand side in the vector *b* if the parameter *ipar*(13) is positive, and the restarted version of the FGMRES method can not be run. If you want to keep the right-hand side, you must save it in a different memory location before the first call to the `dfgmres_get` routine with a positive *ipar*(13).

The parameter *RCI_request* gives information about the task completion and requests results of certain operations that the solver requires.

The lengths of all the vectors must be defined in a previous call to the `dfgmres_init` routine.

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>x</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the initial approximation to the solution vector.
<i>b</i>	DOUBLE PRECISION. Array of size <i>n</i> . Contains the right-hand side vector.
<i>tmp</i>	DOUBLE PRECISION. Array of size $((2*ipar(15)+1)*n + ipar(15)*ipar(15)+9)/2 + 1$. Refer to the FGMRES Common Parameters .

Output Parameters

<i>RCI_request</i>	INTEGER. Informs about result of work of the routine.
<i>ipar</i>	INTEGER. Array of size 128. Refer to the FGMRES Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size 128. Refer to the FGMRES Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size $((2*ipar(15)+1)*n + ipar(15)*ipar(15)+9)/2 + 1$. Refer to the FGMRES Common Parameters .

Return Values

<i>RCI_request</i> =0	Indicates that the task completed normally and the solution is found and stored in the vector <i>x</i> . This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the <i>RCI_request</i> = 2 or 4.
<i>RCI_request</i> =-1	Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if you request both tests.
<i>RCI_request</i> = -10	Indicates that the routine was interrupted because of an attempt to divide by zero. Usually this happens if the matrix is degenerate or almost degenerate. However, it may happen if the parameter <i>dpar</i> is altered, or if the method is not stopped when the solution is found.
<i>RCI_request</i> = -11	Indicates that the routine was interrupted because it entered an infinite cycle. Usually this happens because the values <i>ipar</i> (8), <i>ipar</i> (9), <i>ipar</i> (10) were altered outside of the routine, or the <code>dfgmres_check</code> routine was not called.

<i>RCI_request</i> = -12	Indicates that the routine was interrupted because errors were found in the method parameters. Usually this happens if the parameters <i>ipar</i> and <i>dpar</i> were altered by mistake outside the routine.
<i>RCI_request</i> = 1	Indicates that you must multiply the matrix by <i>tmp(ipar(22))</i> , put the result in the <i>tmp(ipar(23))</i> , and return the control back to the routine <i>dfgmres</i> .
<i>RCI_request</i> = 2	Indicates that you must perform the stopping tests. If they fail, return control to the <i>dfgmres</i> routine. Otherwise, the FGMRES solution is found, and you can run the <i>fgmres_get</i> routine to update the computed solution in the vector <i>x</i> .
<i>RCI_request</i> = 3	Indicates that you must apply the inverse preconditioner to <i>ipar(22)</i> , put the result in the <i>ipar(23)</i> , and return the control back to the routine <i>dfgmres</i> .
<i>RCI_request</i> = 4	Indicates that you must check the norm of the currently generated vector. If it is not zero within the computational/rounding errors, return control to the <i>dfgmres</i> routine. Otherwise, the FGMRES solution is found, and you can run the <i>dfgmres_get</i> routine to update the computed solution in the vector <i>x</i> .

dfgmres_get

Retrieves the number of the current iteration and updates the solution.

Syntax

Fortran:

```
dfgmres_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)
```

C:

```
void dfgmres_get (MKL_INT *n, double *x, double *b, MKL_INT *RCI_request, MKL_INT *ipar, double *dpar, double *tmp, MKL_INT *itercount);
```

Include Files

- Fortran: *mkl_rci.fi*
- Fortran 90: *mkl_rci.f90*
- C: *mkl_rci.h*

Description

The routine *dfgmres_get* retrieves the current iteration number of the solution process and updates the solution according to the computations performed by the *dfgmres* routine. To retrieve the current iteration number only, set the parameter *ipar(13) = -1* beforehand. Normally, you should do this before proceeding further with the computations. If the intermediate solution is needed, the method parameters must be set properly. For details see [FGMRES Common Parameters](#) and the Iterative Sparse Solver code examples in the Intel MKL installation directory:

- Fortran examples: *examples/solverf/source*

- C examples: examples/solverc/source

Input Parameters

<i>n</i>	INTEGER. Sets the size of the problem.
<i>ipar</i>	INTEGER. Array of size 128. Refer to the FGMRES Common Parameters .
<i>dpar</i>	DOUBLE PRECISION. Array of size 128. Refer to the FGMRES Common Parameters .
<i>tmp</i>	DOUBLE PRECISION. Array of size ((2* <i>ipar</i> (15)+1)* <i>n</i> + <i>ipar</i> (15)* <i>ipar</i> (15)+9)/2 + 1). Refer to the FGMRES Common Parameters .

Output Parameters

<i>x</i>	DOUBLE PRECISION. Array of size <i>n</i> . If <i>ipar</i> (13) = 0, it contains the updated approximation to the solution according to the computations done in <code>dfgmres</code> routine. Otherwise, it is not changed.
<i>b</i>	DOUBLE PRECISION. Array of size <i>n</i> . If <i>ipar</i> (13) > 0, it contains the updated approximation to the solution according to the computations done in <code>dfgmres</code> routine. Otherwise, it is not changed.
<i>RCI_request</i>	INTEGER. Gives information about result of the routine.
<i>itercount</i>	INTEGER. Contains the value of the current iteration number.

Return Values

<i>RCI_request</i> = 0	Indicates that the task completed normally.
<i>RCI_request</i> = -12	Indicates that the routine was interrupted because errors were found in the routine parameters. Usually this happens if the parameters <i>ipar</i> and <i>dpar</i> were altered by mistake outside of the routine.
<i>RCI_request</i> = -10000	Indicates that the routine failed to complete the task.

Implementation Details

Several aspects of the Intel MKL RCI ISS interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, include one of the Intel MKL RCI ISS language-specific header files.

The C-language header file defines these function prototypes:

```
void dcg_init(int *n, double *x, double *b, int *rci_request, int *ipar, double dpar,
double *tmp);

void dcg_check(int *n, double *x, double *b, int *rci_request, int *ipar, double dpar,
double *tmp);

void dcg(int *n, double *x, double *b, int *rci_request, int *ipar, double dpar, double
*tmp);

void dcg_get(int *n, double *x, double *b, int *rci_request, int *ipar, double dpar,
double *tmp, int *itercount);
```

```

void dcgmrhs_init(int *n, double *x, int *nRhs, double *b, int *method, int
*rcl_request, int *ipar, double dpar, double *tmp);
void dcgmrhs_check(int *n, double *x, int *nRhs, double *b, int *rcl_request, int
*ipar, double dpar, double *tmp);
void dcgmrhs(int *n, double *x, int *nRhs, double *b, int *rcl_request, int *ipar,
double dpar, double *tmp);
void dcgmrhs_get(int *n, double *x, int *nRhs, double *b, int *rcl_request, int *ipar,
double dpar, double *tmp, int *itercount);

void dfgmres_init(int *n, double *x, double *b, int *rcl_request, int *ipar, double
dpar, double *tmp);
void dfgmres_check(int *n, double *x, double *b, int *rcl_request, int *ipar, double
dpar, double *tmp);
void dfgmres(int *n, double *x, double *b, int *rcl_request, int *ipar, double dpar,
double *tmp);
void dfgmres_get(int *n, double *x, double *b, int *rcl_request, int *ipar, double
dpar, double *tmp, int *itercount);

```

NOTE

Intel MKL does not support the RCI ISS interface unless you include the language-specific header file.

Preconditioners based on Incomplete LU Factorization Technique

Preconditioners, or accelerators are used to accelerate an iterative solution process. In some cases, their use can reduce the number of iterations dramatically and thus lead to better solver performance. Although the terms *preconditioner* and *accelerator* are synonyms, hereafter only *preconditioner* is used.

Intel MKL provides two preconditioners, ILU0 and ILUT, for sparse matrices presented in the format accepted in the Intel MKL direct sparse solvers (three-array variation of the CSR storage format described in [Sparse Matrix Storage Format](#)). The algorithms used are described in [Saad03].

The ILU0 preconditioner is based on a well-known factorization of the original matrix into a product of two triangular matrices: lower and upper triangular matrices. Usually, such decomposition leads to some fill-in in the resulting matrix structure in comparison with the original matrix. The distinctive feature of the ILU0 preconditioner is that it preserves the structure of the original matrix in the result.

Unlike the ILU0 preconditioner, the ILUT preconditioner preserves some resulting fill-in in the preconditioner matrix structure. The distinctive feature of the ILUT algorithm is that it calculates each element of the preconditioner and saves each one if it satisfies two conditions simultaneously: its value is greater than the product of the given tolerance and matrix row norm, and its value is in the given bandwidth of the resulting preconditioner matrix.

Both ILU0 and ILUT preconditioners can apply to any non-degenerate matrix. They can be used alone or together with the Intel MKL RCI FGMRES solver (see [Sparse Solver Routines](#)). Avoid using these preconditioners with MKL RCI CG solver because in general, they produce a non-symmetric resulting matrix even if the original matrix is symmetric. Usually, an inverse of the preconditioner is required in this case. To do this the Intel MKL triangular solver routine `mkl_dcsrtrsv` must be applied twice: for the lower triangular part of the preconditioner, and then for its upper triangular part.

NOTE

Although ILU0 and ILUT preconditioners apply to any non-degenerate matrix, in some cases the algorithm may fail to ensure successful termination and the required result. Whether or not the preconditioner produces an acceptable result can only be determined in practice.

A preconditioner may increase the number of iterations for an arbitrary case of the system and the initial solution, and even ruin the convergence. It is your responsibility as a user to choose a suitable preconditioner.

General Scheme of Using ILUT and RCI FGMRES Routines

The general scheme for use is the same for both preconditioners. Some differences exist in the calling parameters of the preconditioners and in the subsequent call of two triangular solvers. You can see all these differences in the code examples for both preconditioners (`dcsrilu*.f` and `dcsrilu*.c`) in the examples folder of the Intel MKL installation directory:

- Fortran examples: `examples/solverf/source`
- C examples: `examples/solverc/source`

The following pseudocode shows the general scheme of using the ILUT preconditioner in the RCI FGMRES context.

...

generate matrix `A`

generate preconditioner `C` (optional)

```

call dfgmres_init(n, x, b, RCI_request, ipar, dpar, tmp)
    change parameters in ipar, dpar if necessary
    call dcsrilut(n, a, ia, ja, bilut, ibilut, jbilut, tol, maxfil, ipar, dpar, ierr)
    call dfgmres_check(n, x, b, RCI_request, ipar, dpar, tmp)
1   call dfgmres(n, x, b, RCI_request, ipar, dpar, tmp)
    if (RCI_request.eq.1) then
        multiply the matrix A by tmp(ipar(22)) and put the result in tmp(ipar(23))
c    proceed with FGMRES iterations
        goto 1
    endif
    if (RCI_request.eq.2) then
        do the stopping test
        if (test not passed) then
            c proceed with FGMRES iterations
                go to 1
            else
                c stop FGMRES iterations.
                    goto 2
            endif
            if (RCI_request.eq.3) then
                c Below, trvec is an intermediate vector of length at least n
                c Here is the recommended use of the result produced by the ILUT routine.
                c via standard Intel MKL Sparse Blas solver routine mkl_dcsrtrsv.
            endif
        endif
    endif

```

```

call mkl_dcsrtrsv('L','N','U', n, bilut, ibilut, jbilut, tmp(ipar(22)),trvec)
call mkl_dcsrtrsv('U','N','N', n, bilut, ibilut, jbilut, trvec, tmp(ipar(23)))
c proceed with FGMRES iterations

    goto 1
endif
if (RCI_request.eq.4) then
    check the norm of the next orthogonal vector, it is contained in dpar(7)
    if (the norm is not zero up to rounding/computational errors) then
c proceed with FGMRES iterations

    goto 1
else
c stop FGMRES iterations

    goto 2
endif
endif
2 call dfgmres_get(n, x, b, RCI_request, ipar, dpar, tmp, itercount)
current iteration number is in itercount
the computed approximation is in the array x

```

ILU0 and ILUT Preconditioners Interface Description

The concepts required to understand the use of the Intel MKL preconditioner routines are discussed in the [Appendix A Linear Solvers Basics](#).

In this section the FORTRAN style notations are used. All types refer to the standard Fortran types, INTEGER, and DOUBLE PRECISION.

C and C++ programmers must refer to the section [Calling Sparse Solver and Preconditioner Routines from C/C++](#) for information on mapping Fortran types to C/C++ types.

User Data Arrays

The preconditioner routines take arrays of user data as input. To minimize storage requirements and improve overall run-time efficiency, the Intel MKL preconditioner routines do not make copies of the user input arrays.

Common Parameters

Some parameters of the preconditioners are common with the [FGMRES Common Parameters](#). The routine `dfgmres_init` specifies their default and initial values. However, some parameters can be redefined with other values. These parameters are listed below.

For the ILU0 preconditioner:

`ipar(2)` - specifies the destination of error messages generated by the ILU0 routine. The default value 6 means that all error messages are displayed on the screen. Otherwise routine creates a log file called `MKL_PREC_log.txt` and writes error messages to it. Note if the parameter `ipar(6)` is set to 0, then error messages are not generated at all.

ipar(6) - specifies whether error messages are generated. If its value is not equal to 0, the ILU0 routine returns error messages as specified by the parameter *ipar(2)*. Otherwise, the routine does not generate error messages at all, but returns a negative value for the parameter *ierr*. The default value is 1.

For the ILUT preconditioner:

ipar(2) - specifies the destination of error messages generated by the ILUT routine. The default value 6 means that all messages are displayed on the screen. Otherwise routine creates a log file called MKL_PREC_log.txt and writes error messages to it. Note if the parameter *ipar(6)* is set to 0, then error messages are not generated at all.

ipar(6) - specifies whether error messages are generated. If its value is not equal to 0, the ILUT routine returns error messages as specified by the parameter *ipar(2)*. Otherwise, the routine does not generate error messages at all, but returns a negative value for the parameter *ierr*. The default value is 1.

ipar(7) - if its value is greater than 0, the ILUT routine generates warning messages as specified by the parameter *ipar(2)* and continues calculations. If its value is equal to 0, the routine returns a positive value of the parameter *ierr*. If its value is less than 0, the routine generates a warning message as specified by the parameter *ipar(2)* and returns a positive value of the parameter *ierr*. The default value is 1.

dcsrilo0

ILU0 preconditioner based on incomplete LU factorization of a sparse matrix.

Syntax

Fortran:

```
call dcsrilo0(n, a, ia, ja, bilu0, ipar, dpar, ierr)
```

C:

```
void dcsrilo0 (MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , double *bilu0 ,
MKL_INT *ipar , double *dpar , MKL_INT *ierr );
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The routine dcsrilo0 computes a preconditioner B [Saad03] of a given sparse matrix A stored in the format accepted in the direct sparse solvers:

$A \sim B = L^* U$, where L is a lower triangular matrix with a unit diagonal, U is an upper triangular matrix with a non-unit diagonal, and the portrait of the original matrix A is used to store the incomplete factors L and U .

CAUTION

This routine supports only one-based indexing of the array parameters, regardless of whether the Fortran or C interface is used.

Input Parameters

<i>n</i>	INTEGER. Size (number of rows or columns) of the original square <i>n</i> -by- <i>n</i> matrix <i>A</i> .
----------	---

<i>a</i>	DOUBLE PRECISION. Array containing the set of elements of the matrix <i>A</i> . Its length is equal to the number of non-zero elements in the matrix <i>A</i> . Refer to the <i>values</i> array description in the Sparse Matrix Storage Format for more details.
<i>ia</i>	INTEGER. Array of size $(n+1)$ containing begin indices of rows of the matrix <i>A</i> such that <i>ia</i> (<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> (<i>n</i> +1) is equal to the number of non-zero elements in the matrix <i>A</i> , plus one. Refer to the <i>rowIndex</i> array description in the Sparse Matrix Storage Format for more details.
<i>ja</i>	INTEGER. Array containing the column indices for each non-zero element of the matrix <i>A</i> . It is important that the indices are in increasing order per row. The matrix size is equal to the size of the array <i>a</i> . Refer to the <i>columns</i> array description in the Sparse Matrix Storage Format for more details.

CAUTION

If column indices are not stored in ascending order for each row of matrix, the result of the routine might not be correct.

<i>ipar</i>	INTEGER. Array of size 128. This parameter specifies the integer set of data for both the ILU0 and RCI FGMRES computations. Refer to the <i>ipar</i> array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries that are specific to ILU0 are listed below.
<i>ipar</i> (31)	specifies how the routine operates when a zero diagonal element occurs during calculation. If this parameter is set to 0 (the default value set by the routine dfgmres_init), then the calculations are stopped and the routine returns a non-zero error value. Otherwise, the diagonal element is set to the value of <i>dpar</i> (32) and the calculations continue.

NOTE

You can declare the *ipar* array with a size of 32. However, for future compatibility you must declare the array *ipar* with length 128.

<i>dpar</i>	DOUBLE PRECISION. Array of size 128. This parameter specifies the double precision set of data for both the ILU0 and RCI FGMRES computations. Refer to the <i>dpar</i> array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries specific to ILU0 are listed below.
<i>dpar</i> (31)	specifies a small value, which is compared with the computed diagonal elements. When <i>ipar</i> (31) is not 0, then diagonal elements less than <i>dpar</i> (31) are set to <i>dpar</i> (32). The default value is 1.0D-16.

NOTE

This parameter can be set to the negative value, because the calculation uses its absolute value.

If this parameter is set to 0, the comparison with the diagonal element is not performed.

dpar(32)

specifies the value that is assigned to the diagonal element if its value is less than *dpar*(31) (see above). The default value is 1.0D-10.

NOTE

You can declare the *dpar* array with a size of 32. However, for future compatibility you must declare the array *dpar* with length 128.

Output Parameters

bilu0

DOUBLE PRECISION. Array *B* containing non-zero elements of the resulting preconditioning matrix *B*, stored in the format accepted in direct sparse solvers. Its size is equal to the number of non-zero elements in the matrix *A*. Refer to the *values* array description in the [Sparse Matrix Storage Format](#) section for more details.

ierr

INTEGER. Error flag, gives information about the routine completion.

NOTE

To present the resulting preconditioning matrix in the CSR3 format the arrays *ia* (row indices) and *ja* (column indices) of the input matrix must be used.

Return Values

ierr=0

Indicates that the task completed normally.

ierr=-101

Indicates that the routine was interrupted and that error occurred: at least one diagonal element is omitted from the matrix in CSR3 format (see [Sparse Matrix Storage Format](#)).

ierr=-102

Indicates that the routine was interrupted because the matrix contains a diagonal element with the value of zero.

ierr=-103

Indicates that the routine was interrupted because the matrix contains a diagonal element which is so small that it could cause an overflow, or that it would cause a bad approximation to ILU0.

ierr=-104

Indicates that the routine was interrupted because the memory is insufficient for the internal work array.

ierr=-105

Indicates that the routine was interrupted because the input matrix size *n* is less than or equal to 0.

ierr=-106

Indicates that the routine was interrupted because the column indices *ja* are not in the ascending order.

Interfaces

FORTRAN 77 and Fortran 95:

```
SUBROUTINE dcsrilu0 (n, a, ia, ja, bilu0, ipar, dpar, ierr)
INTEGER n, ierr, ipar(128)
INTEGER ia(*), ja(*)
DOUBLE PRECISION a(*), bilu0(*), dpar(128)
```

C:

```
void dcsrilu0 (int *n, double *a, int *ia, int *ja, double *bilu0, int *ipar, double *dpar, int *ierr);
```

dcsrilut

ILUT preconditioner based on the incomplete LU factorization with a threshold of a sparse matrix.

Syntax

Fortran:

```
call dcsrilut(n, a, ia, ja, bilut, ibilut, jibilut, tol, maxfil, ipar, dpar, ierr)
```

C:

```
void dcsrilut (MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, double *bilut, MKL_INT
*ibilut, MKL_INT *jibilut, double *tol, MKL_INT *maxfil, MKL_INT *ipar, double *dpar,
MKL_INT *ierr);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The routine `dcsrilut` computes a preconditioner B [Saad03] of a given sparse matrix A stored in the format accepted in the direct sparse solvers:

$A \sim B = L^* U$, where L is a lower triangular matrix with unit diagonal and U is an upper triangular matrix with non-unit diagonal.

The following threshold criteria are used to generate the incomplete factors L and U :

- 1) the resulting entry must be greater than the matrix current row norm multiplied by the parameter `tol`, and
- 2) the number of the non-zero elements in each row of the resulting L and U factors must not be greater than the value of the parameter `maxfil`.

CAUTION

This routine supports only one-based indexing of the array parameters, regardless of whether the Fortran or C interface is used.

Input Parameters

<i>n</i>	INTEGER. Size (number of rows or columns) of the original square <i>n</i> -by- <i>n</i> matrix <i>A</i> .
<i>a</i>	DOUBLE PRECISION. Array containing all non-zero elements of the matrix <i>A</i> . The length of the array is equal to their number. Refer to <i>values</i> array description in the Sparse Matrix Storage Format section for more details.
<i>ia</i>	INTEGER. Array of size (<i>n</i> +1) containing indices of non-zero elements in the array <i>a</i> . <i>ia</i> (<i>i</i>) is the index of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> (<i>n</i> +1) is equal to the number of non-zeros in the matrix <i>A</i> , plus one. Refer to the <i>rowIndex</i> array description in the Sparse Matrix Storage Format for more details.
<i>ja</i>	INTEGER. Array of size equal to the size of the array <i>a</i> . This array contains the column numbers for each non-zero element of the matrix <i>A</i> . It is important that the indices are in increasing order per row. Refer to the <i>columns</i> array description in the Sparse Matrix Storage Format for more details.

CAUTION

If column indices are not stored in ascending order for each row of matrix, the result of the routine might not be correct.

<i>tol</i>	DOUBLE PRECISION. Tolerance for threshold criterion for the resulting entries of the preconditioner.
<i>maxfil</i>	INTEGER. Maximum fill-in, which is half of the preconditioner bandwidth. The number of non-zero elements in the rows of the preconditioner can not exceed (2* <i>maxfil</i> +1).
<i>ipar</i>	INTEGER array of size 128. This parameter is used to specify the integer set of data for both the ILUT and RCI FGMRES computations. Refer to the <i>ipar</i> array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries specific to ILUT are listed below.

ipar(31)

specifies how the routine operates if the value of the computed diagonal element is less than the current matrix row norm multiplied by the value of the parameter *tol*. If *ipar*(31) = 0, then the calculation is stopped and the routine returns non-zero error value. Otherwise, the value of the diagonal element is set to a value determined by *dpar*(31) (see its description below), and the calculations continue.

NOTE

There is no default value for *ipar*(31) even if the preconditioner is used within the RCI ISS context. Always set the value of this entry.

NOTE

You must declare the array `ipar` with length 128. While defining the array in the code as `INTEGER ipar(31)` works, there is no guarantee of future compatibility with Intel MKL.

dpar

DOUBLE PRECISION array of size 128. This parameter specifies the double precision set of data for both ILUT and RCI FGMRES computations. Refer to the `dpar` array description in the [FGMRES Common Parameters](#) for more details on FGMRES parameter entries. The entries that are specific to ILUT are listed below.

dpar(31)

used to adjust the value of small diagonal elements. Diagonal elements with a value less than the current matrix row norm multiplied by `tol` are replaced with the value of `dpar(31)` multiplied by the matrix row norm.

NOTE

There is no default value for `dpar(31)` entry even if the preconditioner is used within RCI ISS context. Always set the value of this entry.

NOTE

You must declare the array `dpar` with length 128. While defining the array in the code as DOUBLE PRECISION `ipar(31)` works, there is no guarantee of future compatibility with Intel MKL.

Output Parameters

bilut

DOUBLE PRECISION. Array containing non-zero elements of the resulting preconditioning matrix B , stored in the format accepted in the direct sparse solvers. Refer to the `values` array description in the [Sparse Matrix Storage Format](#) for more details. The size of the array is equal to $(2*maxfil+1)*n-maxfil*(maxfil+1)+1$.

NOTE

Provide enough memory for this array before calling the routine. Otherwise, the routine may fail to complete successfully with a correct result.

ibilut

INTEGER. Array of size $(n+1)$ containing indices of non-zero elements in the array `bilut`. `ibilut(i)` is the index of the first non-zero element from the row i . The value of the last element `ibilut(n+1)` is equal to the number of non-zeros in the matrix B , plus one. Refer to the `rowIndex` array description in the [Sparse Matrix Storage Format](#) for more details.

<i>jbilut</i>	INTEGER. Array, its size is equal to the size of the array <i>bilut</i> . This array contains the column numbers for each non-zero element of the matrix <i>B</i> . Refer to the <i>columns</i> array description in the Sparse Matrix Storage Format for more details.
<i>ierr</i>	INTEGER. Error flag, informs about the routine completion.

Return Values

<i>ierr</i> =0	Indicates that the task completed normally.
<i>ierr</i> =-101	Indicates that the routine was interrupted because of an error: the number of elements in some matrix row specified in the sparse format is equal to or less than 0.
<i>ierr</i> =-102	Indicates that the routine was interrupted because the value of the computed diagonal element is less than the product of the given tolerance and the current matrix row norm, and it cannot be replaced as <i>dpar</i> (31)=0.
<i>ierr</i> =-103	Indicates that the routine was interrupted because the element <i>ia</i> (<i>i</i> +1) is less than or equal to the element <i>ia</i> (<i>i</i>) (see Sparse Matrix Storage Format).
<i>ierr</i> =-104	Indicates that the routine was interrupted because the memory is insufficient for the internal work arrays.
<i>ierr</i> =-105	Indicates that the routine was interrupted because the input value of <i>maxfil</i> is less than 0.
<i>ierr</i> =-106	Indicates that the routine was interrupted because the size <i>n</i> of the input matrix is less than 0.
<i>ierr</i> =-107	Indicates that the routine was interrupted because an element of the array <i>ja</i> is less than 1, or greater than <i>n</i> (see Sparse Matrix Storage Format).
<i>ierr</i> =101	The value of <i>maxfil</i> is greater than or equal to <i>n</i> . The calculation is performed with the value of <i>maxfil</i> set to (<i>n</i> -1).
<i>ierr</i> =102	The value of <i>tol</i> is less than 0. The calculation is performed with the value of the parameter set to (- <i>tol</i>).
<i>ierr</i> =103	The absolute value of <i>tol</i> is greater than value of <i>dpar</i> (31); it can result in instability of the calculation.
<i>ierr</i> =104	The value of <i>dpar</i> (31) is equal to 0. It can cause calculations to fail.

Interfaces

FORTRAN 77 and Fortran 95:

```
SUBROUTINE dcsrilut (n, a, ia, ja, bilut, ibilut, jbilut, tol, maxfil, ipar, dpar, ierr)
INTEGER n, ierr, ipar(*), maxfil
INTEGER ia(*), ja(*), ibilut(*), jbilut(*)
DOUBLE PRECISION a(*), bilut(*), dpar(*), tol
```

C:

```
void dcsrilut (int *n, double *a, int *ia, int *ja, double *bilut, int *ibilut, int *jbilut, double
*tol, int *maxfil, int *ipar, double *dpar, int *ierr);
```

Sparse Matrix Checker Routines

Intel MKL provides a sparse matrix checker so that you can find errors in the storage of sparse matrices before calling Intel MKL PARDISO, DSS, or Sparse BLAS routines.

[sparse_matrix_checker](#)

Checks correctness of sparse matrix.

Syntax

Fortran:

```
error = sparse_matrix_checker (handle)
```

C:

```
MKL_INT sparse_matrix_checker (sparse_struct* handle);
```

Include Files

- Fortran: `mkl_sparse_handle.fi`
- Fortran 90: `mkl_sparse_handle.f90`
- C: `mkl_sparse_handle.h`

Description

The `sparse_matrix_checker` routine checks a user-defined array used to store a sparse matrix in order to detect issues which could cause problems in routines that require sparse input matrices, such as Intel MKL PARDISO, DSS, or Sparse BLAS.

Input Parameters

`handle`

FORTRAN 77: INTEGER*8

Fortran 90: TYPE (SPARSE_STRUCT), INTENT(INOUT)

Pointer to the data structure describing the sparse array to check.

Return Values

The routine returns a value `error`. Additionally, the `check_result` parameter returns information about where the error occurred, which can be used when `message_level` is `MKL_NO_PRINT`.

Sparse Matrix Checker Error Values

error value	Meaning	Location
MKL_SPARSE_CHECKER_SUCCESS	The input array successfully passed all checks.	
MKL_SPARSE_CHECKER_NON_MONOTONIC	The input array is not 0 or 1 based (Fortran: <code>ia(1)</code> , C: <code>ia[0]</code>) or elements of <code>ia</code> are not in non-decreasing order as required.	Fortran:
		<code>ia(i + 1)</code> and <code>ia(i + 2)</code> are incompatible. <code>check_result(1) = i</code> <code>check_result(2) = ia(i + 1)</code> <code>check_result(3) = ia(i + 2)</code>
MKL_SPARSE_CHECKER_OUT_OF_BOUNDS	The value of the <code>ja</code> array is lower than the number of the first column or greater than the number of the last column.	C:
		<code>ia[i]</code> and <code>ia[i + 1]</code> are incompatible. <code>check_result[0] = i</code> <code>check_result[1] = ia[i]</code> <code>check_result[2] = ia[i + 1]</code>
MKL_SPARSE_CHECKER_NONTRIANGULAR	The <code>matrix_structure</code> parameter is <code>MKL_UPPER_TRIANGULAR</code> and both <code>ia</code> and <code>ja</code> are not upper triangular, or the <code>matrix_structure</code> parameter is <code>MKL_LOWER_TRIANGULAR</code> and both <code>ia</code> and <code>ja</code> are not lower triangular	Fortran:
		<code>ia(i + 1)</code> and <code>ja(j + 1)</code> are incompatible. <code>check_result(1) = i</code> <code>check_result(2) = ia(i + 1) = j</code> <code>check_result(3) = ja(j + 1)</code>
		C:
		<code>ia[i]</code> and <code>ja[j]</code> are incompatible. <code>check_result[0] = i</code> <code>check_result[1] = ia[i] = j</code> <code>check_result[2] = ja[j]</code>

error value	Meaning	Location
MKL_SPARSE_CHECKER_NONORD	The elements of the ja array are not in non-decreasing order in each row as required.	<p>Fortran:</p> <p><i>ja(j + 1)</i> and <i>ja(j + 2)</i> are incompatible.</p> <pre>check_result(1) = j check_result(2) = ja(j + 1) check_result(3) = ja(j + 2)</pre> <p>C:</p> <p><i>ia[i]</i> and <i>ia[i + 1]</i> are incompatible.</p> <pre>check_result[0] = j check_result[1] = ja[j] check_result[2] = ja[j + 1]</pre>

See Also

[sparse_matrix_init](#) Initializes handle for sparse matrix checker.

[Intel MKL PARDISO - Parallel Direct Sparse Solver Interface](#)

[Sparse BLAS Level 2 and Level 3 Routines](#)

[Sparse Matrix Storage Formats](#)

[sparse_matrix_init](#)

Initializes handle for sparse matrix checker.

Syntax

Fortran:

```
call sparse_matrix_init (handle)
```

C:

```
void sparse_matrix_init (sparse_struct* handle);
```

Include Files

- Fortran: `mkl_sparse_handle.fi`
- Fortran 90: `mkl_sparse_handle.f90`
- C: `mkl_sparse_handle.h`

Description

The `sparse_matrix_init` routine initializes the handle for the `sparse_matrix_checker` routine. The `handle` variable contains this data:

Description of `sparse_matrix_checker` handle Data

Field	Type	Possible Values	Meaning
n	Fortran: INTEGER C: MKL_INT		Order of the matrix stored in sparse array.

Field	Type	Possible Values	Meaning
csr_ia	Fortran: INTEGER (C_INTPTR_T) C: MKL_INT*	Pointer to <i>ia</i> array for matrix_format = MKL_CSR	
csr_ja	Fortran: INTEGER (C_INTPTR_T) C: MKL_INT*	Pointer to <i>ja</i> array for matrix_format = MKL_CSR	
check_result(3)	Fortran: INTEGER (KIND=4) C: MKL_INT	See Sparse Matrix Checker Error Values for a description of the values returned in <i>check_result</i> .	Indicates location of problem in array when message_level = MKL_NO_PRINT.
indexing	Fortran: INTEGER (KIND=4) C: sparse_matrix_indexing	MKL_ZERO_BASED MKL_ONE_BASED	Indexing style used in array.
matrix_structure	Fortran: INTEGER (KIND=4) C: sparse_matrix_structures	MKL_GENERAL_STRUCTURE MKL_UPPER_TRIANGULAR MKL_LOWER_TRIANGULAR MKL_STRUCTURAL_SYMMETRIC	Type of sparse matrix stored in array.
matrix_format	Fortran: INTEGER (KIND=4) C: sparse_matrix_formats	MKL_CSR	Format of array used for sparse matrix storage.
message_level	Fortran: INTEGER (KIND=4) C: sparse_matrix_message_levels	MKL_NO_PRINT MKL_PRINT	Determines whether or not feedback is provided on the screen.
print_style	Fortran: INTEGER (KIND=4) C: sparse_matrix_print_styles	MKL_C_STYLE MKL_FORTRAN_STYLE	Determines style of messages when message_level = MKL_PRINT.

Input Parameters

<i>handle</i>	FORTRAN 77: INTEGER*8 Fortran 90: TYPE (SPARSE_STRUCT), INTENT(INOUT)
	Pointer to the data structure describing the sparse array to check.

Output Parameters

<i>handle</i>	Pointer to the initialized data structure.
---------------	--

See Also

[sparse_matrix_checker](#) Checks correctness of sparse matrix.

[Intel MKL PARDISO - Parallel Direct Sparse Solver Interface](#)

[Sparse BLAS Level 2 and Level 3 Routines](#)

[Sparse Matrix Storage Formats](#)

Calling Sparse Solver and Preconditioner Routines from C and C++

All of the Intel MKL sparse solver and preconditioner routines are designed to be called easily from FORTRAN 77 or Fortran 90. However, any of these routines can be invoked directly from C or C++ if you are familiar with the inter-language calling conventions of your platforms. These conventions include, but are not limited to, the argument passing mechanisms for the language and the platform-specific method of decoration for Fortran external names.

To promote portability, the C header files provide a set of macros intended to hide the inter-language calling conventions and provide an interface to the Intel MKL sparse solver routines that appears natural for C and C++.

For example, consider a hypothetical library routine `foo` that takes a real vector of length `n`, and returns an integer status. Fortran users would access such a function using code such as:

```
INTEGER n, status, foo
REAL x(*)
status = foo(x, n)
```

As noted above, to invoke `foo`, C users need to know what argument passing mechanism the Fortran compiler uses and what, if any, name decoration the Fortran compiler performs when generating the external symbol `foo`.

However, by using the C specific header file, for example `mkl_solver.h`, the invocation of `foo`, within a C program would look like the following:

```
#include "mkl_solver.h"
MKL_INT i, status;
float x[];
status = foo( x, i );
```

Passing Constants as Parameters

One of the key differences between C/C++ and Fortran is the argument passing mechanisms for the languages: Fortran programs pass arguments by reference and C/C++ programs pass arguments by value. In the above example, the header file `mkl_solver.h` attempts to hide this difference by defining a macro `foo`, which takes the address of the appropriate arguments. For example, on the Tru64 UNIX* operating system `mkl_solver.h` defines the macro as follows:

```
#define foo(a,b) foo_((a), &(b))
```

Note how constants are treated when using the macro form of `foo`. `foo(x, 10)` is converted into `foo_(x, &10)`. In a strictly ANSI compliant C compiler, taking the address of a constant is not permitted, so a strictly conforming program would look like:

```
MKL_INT iTen = 10;
float * x;
status = foo( x, iTen );
```

However, some C compilers in a non-ANSI compliant mode enable taking the address of a constant for ease of use with Fortran programs. The form `foo(x, 10)` is acceptable for such compilers.

Extended Eigensolver Routines

The Extended Eigensolver functionality in Intel Math Kernel Library (Intel MKL) is based on the FEAST Eigenvalue Solver 2.0 (<http://www ecs umass edu/~polizzi/feast>) in compliance with a BSD license agreement.

Extended Eigensolver uses the same naming convention as the original FEAST package.

NOTE

Intel MKL only supports the shared memory programming (SMP) version of the eigenvalue solver.

- [The FEAST Algorithm](#) gives a brief description of the algorithm underlying the Extended Eigensolver.
- [Extended Eigensolver Functionality](#) describes the problems that can and cannot be solved with the Extended Eigensolver and how to get the best results from the routines.
- [Extended Eigensolver Interfaces](#) gives a reference for calling Extended Eigensolver routines.

The FEAST Algorithm

The Extended Eigensolver functionality is a set of high-performance numerical routines for solving symmetric standard eigenvalue problems, $Ax = \lambda x$, or generalized symmetric-definite eigenvalue problems, $Ax = \lambda Bx$. It yields all the eigenvalues (λ) and eigenvectors (x) within a given search interval $[\lambda_{\min}, \lambda_{\max}]$. It is based on the FEAST algorithm, an innovative fast and stable numerical algorithm presented in [Polizzi09], which fundamentally differs from the traditional Krylov subspace iteration based techniques (Arnoldi and Lanczos algorithms [Bai00]) or other Davidson-Jacobi techniques [Sleijpen96]. The FEAST algorithm is inspired by the density-matrix representation and contour integration techniques in quantum mechanics.

The FEAST numerical algorithm obtains eigenpair solutions using a numerically efficient contour integration technique. The main computational tasks in the FEAST algorithm consist of solving a few independent linear systems along the contour and solving a reduced eigenvalue problem. Consider a circle centered in the middle of the search interval $[\lambda_{\min}, \lambda_{\max}]$. The numerical integration over the circle in the current version of FEAST is performed using N_e -point Gauss-Legendre quadrature with x_e the e -th Gauss node associated with the weight ω_e . For example, for the case $N_e = 8$:

$$\begin{aligned} (x_1, \omega_1) &= (0.183434642495649, 0.362683783378361), \\ (x_2, \omega_2) &= (-0.183434642495649, 0.362683783378361), \\ (x_3, \omega_3) &= (0.525532409916328, 0.313706645877887), \\ (x_4, \omega_4) &= (-0.525532409916328, 0.313706645877887), \\ (x_5, \omega_5) &= (0.796666477413626, 0.222381034453374), \\ (x_6, \omega_6) &= (-0.796666477413626, 0.222381034453374), \\ (x_7, \omega_7) &= (0.960289856497536, 0.101228536290376), \text{ and} \\ (x_8, \omega_8) &= (-0.960289856497536, 0.101228536290376). \end{aligned}$$

The figure [FEAST Pseudocode](#) shows the basic pseudocode for the FEAST algorithm for the case of real symmetric (left pane) and complex Hermitian (right pane) generalized eigenvalue problems, using N for the size of the system and M for the number of eigenvalues in the search interval (see [Polizzi09]).

NOTE

The pseudocode presents a simplified version of the actual algorithm. Refer to <http://arxiv.org/abs/1302.0432> for an in-depth presentation and mathematical proof of convergence of FEAST.

FEAST Pseudocode

<p><i>A</i>: real symmetric <i>B</i>: symmetric positive definite (SPD) $\Re\{x\}$: real part of <i>x</i></p> <ol style="list-style-type: none"> Select $M_0 > M$ random vectors $Y_{N \times M_0} \in \mathbb{R}^{N \times M_0}$. Set $Q = 0$ with $Q \in \mathbb{R}^{N \times M_0}$; $r = (\lambda_{\max} - \lambda_{\min}) / 2$; For $e = 1, \dots, N_e$ compute $\theta_e = -(\pi / 2)(x_e - 1)$, compute $Z_e = (\lambda_{\max} + \lambda_{\min}) / 2 + r \exp(i\theta_e)$, solve $(Z_e B - A)Q_e = Y$ to obtain $Q_e \in \mathbb{C}^{N \times M_0}$ compute $Q = Q + (\omega_e / 2)\Re[r \exp(i\theta_e)Q_e]$ End Form $A_{Q_{M_0 \times M_0}} = Q^T A Q$ and $B_{Q_{M_0 \times M_0}} = Q^T B Q$ reduce value of M_0 if B_Q is not symmetric positive definite. Solve $A_Q \Phi = \varepsilon B_Q \Phi$ to obtain the M_0 eigenvalue ε_m, and eigenvectors $\Phi_{M_0 \times M_0} \in \mathbb{R}^{M_0 \times M_0}$. Set $\lambda_m = \varepsilon_m$ and compute $X_{N \times M_0} = Q_{N \times M_0} \Phi_{M_0 \times M_0}$. If $\lambda_m \in [\lambda_{\min}, \lambda_{\max}]$, λ_m is an eigenvalue solution and its eigenvector is X_m (the m-th column of X). Check convergence for the trace of the eigenvalues λ_m. If iterative refinement is needed, compute $Y = BX$ and go back to step 2. 	<p><i>A</i>: complex Hermitian <i>B</i>: Hermitian positive definite (HPD)</p> <ol style="list-style-type: none"> Select $M_0 > M$ random vectors $Y_{N \times M_0} \in \mathbb{C}^{N \times M_0}$. Set $Q = 0$ with $Q \in \mathbb{R}^{N \times M_0}$; $r = (\lambda_{\max} - \lambda_{\min}) / 2$; For $e = 1, \dots, N_e$ compute $\theta_e = -(\pi / 2)(x_e - 1)$, compute $Z_e = (\lambda_{\max} + \lambda_{\min}) / 2 + r \exp(i\theta_e)$, solve $(Z_e B - A)Q_e = Y$ to obtain $Q_e \in \mathbb{C}^{N \times M_0}$ solve $(Z_e B - A)^H \hat{Q}_e = Y$ to obtain $\hat{Q}_e \in \mathbb{C}^{N \times M_0}$ $Q = Q - (\omega_e / 4)r(\exp(i\theta_e)Q_e + \exp(-i\theta_e)\hat{Q}_e)$ End Form $A_{Q_{M_0 \times M_0}} = Q^H A Q$ and $B_{Q_{M_0 \times M_0}} = Q^H B Q$ reduce value of M_0 if B_Q is not Hermitian positive definite. Solve $A_Q \Phi = \varepsilon B_Q \Phi$ to obtain the M_0 eigenvalue ε_m, and eigenvectors $\Phi_{M_0 \times M_0} \in \mathbb{C}^{M_0 \times M_0}$. Set $\lambda_m = \varepsilon_m$ and compute $X_{N \times M_0} = Q_{N \times M_0} \Phi_{M_0 \times M_0}$. If $\lambda_m \in [\lambda_{\min}, \lambda_{\max}]$, λ_m is an eigenvalue solution and its eigenvector is X_m (the m-th column of X). Check convergence for the trace of the eigenvalues λ_m. If iterative refinement is needed, compute $Y = BX$ and go back to step 2.
---	---

Extended Eigensolver Functionality

Use Extended Eigensolver to compute all the eigenvalues and eigenvectors within a given search interval.

The eigenvalue problems covered are as follows:

- standard, $Ax = \lambda x$
 - *A* complex Hermitian
 - *A* real symmetric
- generalized, $Ax = \lambda Bx$
 - *A* complex Hermitian, *B* Hermitian positive definite (hpd)
 - *A* real symmetric and *B* real symmetric positive definite (spd)

The Extended Eigensolver functionality offers:

- Real/Complex and Single/Double precisions: double precision is recommended to provide better accuracy of eigenpairs.
- All Extended Eigensolver interfaces are compatible with FORTRAN 77, Fortran 90, and C.
- Reverse communication interfaces (RCI) provide maximum flexibility for specific applications. RCI are independent of matrix format and inner system solvers, so you must provide your own linear system solvers (direct or iterative) and matrix-matrix multiply routines.
- Predefined driver interfaces for dense, LAPACK banded, and sparse (CSR) formats are less flexible but are optimized and easy to use:

- The Extended Eigensolver interfaces for dense matrices are likely to be slower than the comparable LAPACK routines because the FEAST algorithm has a higher computational cost.
- The Extended Eigensolver interfaces for banded matrices support banded LAPACK-type storage.
- The Extended Eigensolver sparse interfaces support compressed sparse row format and use the Intel MKL PARDISO solver.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Parallelism in Extended Eigensolver Routines

How you achieve parallelism in Extended Eigensolver routines depends on which interface you use. Parallelism (via shared memory programming) is not *explicitly* implemented in Extended Eigensolver routines within one node: the inner linear systems are currently solved one after another.

- Using the Extended Eigensolver RCI interfaces, you can achieve parallelism by providing a threaded inner system solver and a matrix-matrix multiplication routine. When using the RCI interfaces, you are responsible for activating the threaded capabilities of your BLAS and LAPACK libraries most likely using the shell variable `OMP_NUM_THREADS`.
- Using the predefined Extended Eigensolver interfaces, parallelism can be implicitly obtained within the shared memory version of BLAS, LAPACK or Intel MKL PARDISO. The shell variable `MKL_NUM_THREADS` can be used for automatically setting the number of threads (cores) for BLAS, LAPACK, and Intel MKL PARDISO.

Achieving Performance With Extended Eigensolver Routines

In order to use the Extended Eigensolver Routines, you need to provide

- the search interval and the size of the subspace M_0 (overestimation of the number of eigenvalues M within a given search interval);
- the system matrix in dense, banded, or sparse CSR format if the Extended Eigensolver predefined interfaces are used, or a high-performance complex direct or iterative system solver and matrix-vector multiplication routine if RCI interfaces are used.

In return, you can expect

- fast convergence with very high accuracy when seeking up to 1000 eigenpairs (in two to four iterations using $M_0 = 1.5M$, and $N_e = 8$ or at most using $N_e = 16$ contour points);
- an extremely robust approach.

The performance of the basic FEAST algorithm depends on a trade-off between the choices of the number of Gauss quadrature points N_e , the size of the subspace M_0 , and the number of outer refinement loops to reach the desired accuracy. In practice you should use $M_0 > 1.5 M$, $N_e = 8$, and at most two refinement loops.

For better performance:

- M_0 should be much smaller than the size of the eigenvalue problem, so that the arithmetic complexity depends mainly on the inner system solver ($O(NM)$ for narrow-banded or sparse systems).
- Parallel scalability performance depends on the shared memory capabilities of the inner system solver (via the shell variable `MKL_NUM_THREADS`).
- For very large sparse and challenging systems, application users should make use of the Extended Eigensolver RCI interfaces with customized highly-efficient iterative systems solvers and preconditioners.

- For the Extended Eigensolver interfaces for banded matrices, the parallel performance scalability is limited.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Extended Eigensolver Interfaces

Extended Eigensolver Naming Conventions

There are two different types of interfaces available in the Extended Eigensolver routines:

- The reverse communication interfaces (RCI):

```
?feast_<matrix type>_rci
```

These interfaces are matrix free format (the interfaces are independent of the matrix data formats). You must provide matrix-vector multiply and direct/iterative linear system solvers for your own explicit or implicit data format.

- The predefined interfaces:

```
?feast_<matrix type><type of eigenvalue problem>
```

are predefined drivers for ?feast reverse communication interface that act on commonly used matrix data storage (dense, banded and compressed sparse row representation), using internal matrix-vector routines and selected inner linear system solvers.

For these interfaces:

- ? indicates the data type of matrix A (and matrix B if any) defined as follows:

s	real, single precision
d	real, double precision
c	complex, single precision
z	complex , double precision

- <matrix type> defined as follows:

Value of <matrix type>	Matrix format	Inner linear system solver used by Extended Eigensolver
sy	(symmetric real)	
he	(Hermitian complex)	Dense LAPACK dense solvers
sb	(symmetric banded real)	Banded-LAPACK Internal banded solver

Value of <matrix type>	Matrix format	Inner linear system solver used by Extended Eigensolver	
hb	(Hermitian banded complex)		
scsr	(symmetric real)		
hcsr	(Hermitian complex)	Compressed sparse row	PARDISO solver
s	(symmetric real)	Reverse communications	
h	(Hermitian complex)	interfaces	User defined

- <type of eigenvalue problem> is:

gv	generalized eigenvalue problem
ev	standard eigenvalue problem

For example, `sfeast_scsrev` is a single-precision routine with a symmetric real matrix stored in sparse compressed-row format for a standard eigenvalue problem, and `zfeast_hrci` is a complex double-precision routine with a Hermitian matrix using the reverse communication interface.

Note that:

- ? can be s or d if a matrix is real symmetric: <matrix type> is sy, sb, or scsr.
- ? can be c or z if a matrix is complex Hermitian: <matrix type> is he, hb, or hcsr.
- ? can be c or z if the Extended Eigensolver RCI interface is used for solving a complex Hermitian problem.
- ? can be s or d if the Extended Eigensolver RCI interface is used for solving a real symmetric problem.

feastinit

Initialize Extended Eigensolver input parameters with default values.

Syntax

Fortran:

```
call feastinit (fpm)
```

C:

```
feastinit (MKL_INT* fpm);
```

Include Files

- Fortran: `mkl.fi`
- C: `mkl.h`

Description

This routine sets all Extended Eigensolver parameters to their default values.

Output Parameters

<code>fpm</code>	INTEGER
------------------	---------

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See [Extended Eigensolver Input Parameters](#) for a complete description of the parameters and their default values.

Extended Eigensolver Input Parameters

The input parameters for Extended Eigensolver routines are contained in an integer array named fpm . To call the Extended Eigensolver interfaces, this array should be initialized using the routine [feastinit](#).

Param	Default	Description
$fpm(1)$	0	Specifies whether Extended Eigensolver routines print runtime status. $fpm(1)=0$ Extended Eigensolver routines do not generate runtime messages at all. $fpm(1)=1$ Extended Eigensolver routines print runtime status to the screen.
$fpm(2)$	8	The number of contour points $N_e = 8$ (see the description of FEAST algorithm). Must be one of {3,4,5,6,8,10,12,16,20,24,32,40,48}.
$fpm(3)$	12	Error trace double precision stopping criteria ε ($\varepsilon = 10^{-fpm(3)}$) .
$fpm(4)$	20	Maximum number of Extended Eigensolver refinement loops allowed. If no convergence is reached within $fpm(4)$ refinement loops, Extended Eigensolver routines return $info=2$.
$fpm(5)$	0	User initial subspace. If $fpm(5)=0$ then Extended Eigensolver routines generate initial subspace, if $fpm(5)=1$ the user supplied initial subspace is used.
$fpm(6)$	0	Extended Eigensolver stopping test. $fpm(6)=0$ Extended Eigensolvers are stopped if this residual stopping test is satisfied: <ul style="list-style-type: none"> • generalized eigenvalue problem: $\max_{i=1:mode} \frac{\ Ax_i - \lambda_i Bx_i\ _1}{\max(E_{\min} , E_{\max}) \ Bx_i\ _1} < \varepsilon$ • standard eigenvalue problem: $\max_{i=1:mode} \frac{\ Ax_i - \lambda_i x_i\ _1}{\max(E_{\min} , E_{\max}) \ x_i\ _1} < \varepsilon$ where $mode$ is the total number of eigenvalues found in the search interval and $\varepsilon = 10^{-fpm(7)}$ for real and complex or $\varepsilon = 10^{-fpm(3)}$ for double precision and double complex.
$fpm(6)=1$		Extended Eigensolvers are stopped if this trace stopping test is satisfied: $\frac{ trace_j - trace_{j-1} }{\max(E_{\min} , E_{\max})} < \varepsilon,$ where $trace_j$ denotes the sum of all eigenvalues found in the search interval $[e_{\min}, e_{\max}]$ at the j -th Extended Eigensolver iteration:

Param	Default	Description
		$\text{trace}_j = \sum_{i=1}^M \lambda_i^{(j)}$.
$fpm(7)$	5	Error trace single precision stopping criteria ($10^{-fpm(7)}$).
$fpm(14)$	0	$fpm(14)=0$ Standard use for Extended Eigensolver routines. $fpm(14)=1$ Non-standard use for Extended Eigensolver routines: return the computed eigenvectors subspace after one single contour integration.
$fpm(30)$ to $fpm(63)$	-	Reserved for future use.
$fpm(64)$	0	Use the Intel MKL PARDISO solver with the user-defined PARDISO <i>iparm</i> array settings.

NOTE

This option can only be used by Extended Eigensolver Predefined Interfaces for Sparse Matrices.

$fpm(64)=0$	Extended Eigensolver routines use the Intel MKL PARDISO default <i>iparm</i> settings defined by calling the <i>pardisoinit</i> subroutine.
$fpm(64)=1$	The values from $fpm(65)$ to $fpm(128)$ correspond to <i>iparm(1)</i> to <i>iparm(64)</i> respectively according to the formula $fpm(64+i) = iparm(i)$ for $i = 1, 2, \dots, 64$.

NOTE

Using the C language, the components of the fpm array start at 0 and stop at 127. Therefore, the components $fpm[j]$ in C (for $j=0, 1, \dots, 127$) must correspond to the components $fpm(i)$ in Fortran (for $i=1, 2, \dots, 128$) specified above (i.e. $fpm[i-1]=fpm(i)$).

Extended Eigensolver Output Details

Errors and warnings encountered during a run of the Extended Eigensolver routines are stored in an integer variable, *info*. If the value of the output *info* parameter is not 0, either an error or warning was encountered. The possible return values for the *info* parameter along with the error code descriptions are given in the following table.

Return Codes for info Parameter

<i>info</i>	Classification	Description
202	Error	Problem with size of the system n ($n \leq 0$)
201	Error	Problem with size of initial subspace m_0 ($m_0 \leq 0$ or $m_0 > n$)
200	Error	Problem with emin, emax ($\text{emin} \geq \text{emax}$)

info	Classification	Description
(100+ <i>i</i>)	Error	Problem with <i>i</i> -th value of the input Extended Eigensolver parameter ($fpm(i)$). Only the parameters in use are checked.
4	Warning	Successful return of only the computed subspace after call with $fpm(14) = 1$
3	Warning	Size of the subspace $m0$ is too small ($m0 < m$)
2	Warning	No Convergence (number of iteration loops $> fpm(4)$)
1	Warning	No eigenvalue found in the search interval. See remark below for further details.
0	Successful exit	
-1	Error	Internal error for allocation memory.
-2	Error	Internal error of the inner system solver. Possible reasons: not enough memory for inner linear system solver or inconsistent input.
-3	Error	Internal error of the reduced eigenvalue solver Possible cause: matrix B may not be positive definite. It can be checked with LAPACK routines, if necessary.
-(100+ <i>i</i>)	Error	Problem with the <i>i</i> -th argument of the Extended Eigensolver interface.

In some extreme cases the return value $info=1$ may indicate that the Extended Eigensolver routine has failed to find the eigenvalues in the search interval. This situation could arise if a very large search interval is used to locate a small and isolated cluster of eigenvalues (i.e. the dimension of the search interval is many orders of magnitude larger than the number of contour points. It is then either recommended to increase the number of contour points $fpm(2)$ or simply rescale more appropriately the search interval. Rescaling means the initial problem of finding all eigenvalues the search interval $[\lambda_{\min}, \lambda_{\max}]$ for the standard eigenvalue problem $A x = \lambda x$ is replaced with the problem of finding all eigenvalues in the search interval $[\lambda_{\min}/t, \lambda_{\max}/t]$ for the standard eigenvalue problem $(A/t) x = (\lambda/t)x$ where t is a scaling factor.

Extended Eigensolver RCI Routines

If you do not require specific linear system solvers or matrix storage schemes, you can skip this section and go directly to [Extended Eigensolver Predefined Interfaces](#).

Extended Eigensolver RCI Interface Description

The Extended Eigensolver RCI interfaces can be used to solve standard or generalized eigenvalue problems, and are independent of the format of the matrices. As mentioned earlier, the Extended Eigensolver algorithm is based on the contour integration techniques of the matrix resolvent $G(\sigma) = (\sigma B - A)^{-1}$ over a circle. For solving a generalized eigenvalue problem, Extended Eigensolver has to perform one or more of the following operations at each contour point denoted below by Z_e :

- Factorize the matrix $(Z_e * B - A)$
- Solve the linear system $(Z_e * B - A)X = Y$ or $(Z_e * B - A)^H X = Y$ with multiple right hand sides, where H means transpose conjugate
- Matrix-matrix multiply $B X = Y$ or $A X = Y$

For solving a standard eigenvalue problem, replace the matrix B with the identity matrix I .

The primary aim of RCI interfaces is to isolate these operations: the linear system solver, factorization of the matrix resolvent at each contour point, and matrix-matrix multiplication. This gives universality to RCI interfaces as they are independent of data structures and the specific implementation of the operations like

matrix-vector multiplication or inner system solvers. However, this approach requires some additional effort when calling the interface. In particular, operations listed above are performed by routines that you supply on data structures that you find most appropriate for the problem at hand.

To initialize an Extended Eigensolver RCI routine, set the job indicator (*ijob*) parameter to the value -1. When the routine requires the results of an operation, it generates a special value of *ijob* to indicate the operation that needs to be performed. The routine also returns *ze*, the coordinate along the complex contour, the values of array *work* or *workc*, and the number of columns to be used. Your subroutine then must perform the operation at the given contour point *ze*, store the results in prescribed array, and return control to the Extended Eigensolver RCI routine.

The following pseudocode shows the general scheme for using the Extended Eigensolver RCI functionality for a real symmetric problem:

NOTE

The `? option` in `?feast` in the pseudocode given above should be replaced by either `s` or `d`, depending on the matrix data type of the eigenvalue system.

The next pseudocode shows the general scheme for using the Extended Eigensolver RCI functionality for a complex Hermitian problem:

NOTE

The `? option in ?feast in the pseudocode given above should be replaced by either c or z, depending on the matrix data type of the eigenvalue system.`

If case (20) can be avoided, performance could be up to twice as fast, and Extended Eigensolver functionality would use half of the memory.

If an iterative solver is used along with a preconditioner, the factorization of the preconditioner could be performed with $ijob = 10$ (and $ijob = 20$ if applicable) for a given value of Z_e , and the associated iterative solve would then be performed with $ijob = 11$ (and $ijob = 21$ if applicable).

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

?feast srcj/?feast hrcj

Extended Eigensolver RCI interface.

Syntax

Fortran:

```
call sfeast_srci (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,  
lambda, q, m, res, info)  
  
call dfeast_srci (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,  
lambda, q, m, res, info)  
  
call cfeast_hrci (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,  
lambda, q, m, res, info)  
  
call zfeast_hrci (ijob, n, ze, work, workc, aq, sq, fpm, epsout, loop, emin, emax, m0,  
lambda, q, m, res, info)
```

C.

```
void sfeast_srci (MKL_INT* ijob, const MKL_INT* n, MKL_Complex8* ze, float* work,
MKL_Complex8* workc, float* aq, float* sq, MKL_INT* fpm, float* epsout, MKL_INT* loop,
const float* emin, const float* emax, MKL_INT* m0, float* lambda, float* q, MKL_INT* m,
float* res, MKL_INT* info);
```

```

void dfeast_srci (MKL_INT* ijob, const MKL_INT* n, MKL_Complex16* ze, double* work,
                  MKL_Complex16* workc, double* aq, double* sq, MKL_INT* fpm, double* epsout, MKL_INT* loop,
                  const double* emin, const double* emax, MKL_INT* m0, double* lambda, double* q,
                  MKL_INT* m, double* res, MKL_INT* info);

void cfeast_hrci (MKL_INT* ijob, const MKL_INT* n, MKL_Complex8* ze, MKL_Complex8* work,
                  MKL_Complex8* workc, MKL_Complex8* aq, MKL_Complex8* sq, MKL_INT* fpm, float* epsout,
                  MKL_INT* loop, const float* emin, const float* emax, MKL_INT* m0, float* lambda,
                  MKL_Complex8* q, MKL_INT* m, float* res, MKL_INT* info);

void zfeast_hrci (MKL_INT* ijob, const MKL_INT* n, MKL_Complex16* ze, MKL_Complex16* work,
                  MKL_Complex16* workc, MKL_Complex16* aq, MKL_Complex16* sq, MKL_INT* fpm,
                  double* epsout, MKL_INT* loop, const double* emin, const double* emax, MKL_INT* m0,
                  double* lambda, MKL_Complex16* q, MKL_INT* m, double* res, MKL_INT* info);

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

Compute eigenvalues as described in [Extended Eigensolver RCI Interface Description](#).

Input Parameters

<i>ijob</i>	INTEGER	Job indicator variable. On entry, a call to ?feast_srci/?feast_hrci with <i>ijob</i> =-1 initializes the eigensolver.
<i>n</i>	INTEGER	Sets the size of the problem.
<i>work</i>	REAL for sfeast_srci DOUBLE PRECISION for dfeast_srci COMPLEX for cfeast_hrci COMPLEX*16 for zfeast_hrci	Workspace array of dimension <i>n</i> by <i>m0</i> .
<i>workc</i>	COMPLEX for sfeast_srci and cfeast_hrci COMPLEX*16 for dfeast_srci and zfeast_hrci	Workspace array of dimension <i>n</i> by <i>m0</i> .
<i>aq</i> , <i>sq</i>	REAL for sfeast_srci DOUBLE PRECISION for dfeast_srci COMPLEX for cfeast_hrci COMPLEX*16 for zfeast_hrci	Workspace arrays of dimension <i>m0</i> by <i>m0</i> .
<i>fpm</i>	INTEGER	

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See [Extended Eigensolver Input Parameters](#) for a complete description of the parameters and their default values.

emin, emax

REAL for `sfeast_srci` and `cfeast_hrci`

DOUBLE PRECISION for `dfeast_srci` and `zfeast_hrci`

The lower and upper bounds of the interval to be searched for eigenvalues; $\text{emin} \leq \text{emax}$.

m0

INTEGER

On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[\text{emin}, \text{emax}]$. If the initial guess is wrong, Extended Eigensolver routines return $\text{info}=3$.

q

REAL for `sfeast_srci`

DOUBLE PRECISION for `dfeast_srci`

COMPLEX for `cfeast_hrci`

COMPLEX*16 for `zfeast_hrci`

On entry, if $fpm(5)=1$, the array $q(n, m)$ contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

ijob

On exit, the parameter carries the status flag that indicates the condition of the return. The status information is divided into three categories:

1. A zero value indicates successful completion of the task.
2. A positive value indicates that the solver requires a matrix-vector multiplication or solving a specific system with a complex coefficient.
3. A negative value indicates successful initiation.

A non-zero value of *ijob* specifically means the following:

- $ijob = 10$ - factorize the complex matrix $Z_e^*B - A$ at a given contour point Z_e and return the control to the `?feast_srci/?feast_hrci` routine where Z_e is a complex number meaning contour point and its value is defined internally in `?feast_srci/?feast_hrci`.
- $ijob = 11$ - solve the complex linear system $(Z_e^*B - A)*y = workc(n, m0)$, put the solution in $workc(n, m0)$ and return the control to the `?feast_srci/?feast_hrci` routine.
- $ijob = 20$ - factorize the complex matrix $(Z_e^*B - A)^H$ at a given contour point Z_e and return the control to the `?feast_srci/?feast_hrci` routine where Z_e is a complex number meaning contour point and its value is defined internally in `?feast_srci/?feast_hrci`.

The symbol X^H means transpose conjugate of matrix X .

- $ijob = 21$ - solve the complex linear system $(Z_e^*B - A)^H*y = workc(n, m0)$, put the solution in $workc(n, m0)$ and return the control to the `?feast_srci/?feast_hrci` routine. The case $ijob=20$ becomes obsolete if the solve can be performed using the factorization computed for $ijob=10$.

The symbol X^H mean transpose conjugate of matrix X .

- $i_{job} = 30$ - multiply matrix A by $q(n, i:j)$, put the result in $work(n, i:j)$, and return the control to the `?feast_srci/?feast_hrci` routine.
 i is $fpm(24)$, and j is $fpm(24) + fpm(25) - 1$.
- $i_{job} = 40$ - multiply matrix B by $q(n, i:j)$, put the result in $work(n, i:j)$ and return the control to the `?feast_srci/?feast_hrci` routine. If a standard eigenvalue problem is solved, just return $work = q$.
 i is $fpm(24)$, and j is $fpm(24) + fpm(25) - 1$.
- $i_{job} = -2$ - rerun the `?feast_srci/?feast_hrci` task with the same parameters.

ze

COMPLEX for `sfeast_srci` and `cfeast_hrci`

COMPLEX*16 for `dfeast_srci` and `zfeast_hrci`

Defines the coordinate along the complex contour. All values of ze are generated by `?feast_srci/?feast_hrci` internally.

fpm

On output, contains coordinates of columns of work array needed for iterative refinement. (See [Extended Eigensolver RCI Interface Description](#).)

$epsout$

REAL for `sfeast_srci` and `cfeast_hrci`

DOUBLE PRECISION for `dfeast_srci` and `zfeast_hrci`

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|emin|, |emax|)$

$loop$

INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

$lambda$

REAL for `sfeast_srci` and `cfeast_hrci`

DOUBLE PRECISION for `dfeast_srci` and `zfeast_hrci`

Array of length m_0 . On output, the first m entries of $lambda$ are eigenvalues found in the interval.

q

On output, q contains all eigenvectors corresponding to $lambda$.

m

INTEGER

The total number of eigenvalues found in the interval $[emin, emax]$: $0 \leq m \leq m_0$.

res

REAL for `sfeast_srci` and `cfeast_hrci`

DOUBLE PRECISION for `dfeast_srci` and `zfeast_hrci`

Array of length m_0 . On exit, the first m components contain the relative residual vector:

- generalized eigenvalue problem:

$$\frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|Bx_i\|_1}$$

- standard eigenvalue problem:

$$\frac{\|Ax_i - \lambda_i x_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|x_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info

INTEGER

If *info*=0, the execution is successful. If *info* ≠ 0, see [Output Eigensolver info Details](#).

Extended Eigensolver Predefined Interfaces

The predefined interfaces include routines for standard and generalized eigenvalue problems, and for dense, banded, and sparse matrices.

Matrix Type	Standard Eigenvalue Problem	Generalized Eigenvalue Problem
Dense	?feast_syev ?feast_heev	?feast_sygv ?feast_hegv
Banded	?feast_sbev ?feast_hbev	?feast_sbqv ?feast_hbqv
Sparse	?feast_scsrev ?feast_hcsrev	?feast_scsvqv ?feast_hcqv

Matrix Storage

The symmetric and Hermitian matrices used in Extended Eigensolvers predefined interfaces can be stored in full, band, and sparse formats.

- In the full storage format (described in [Full Storage](#) in additional detail) you store all elements, all of the elements in the upper triangle of the matrix, or all of the elements in the lower triangle of the matrix.
- In the band storage format (described in [Band storage](#) in additional detail), you store only the elements along a diagonal band of the matrix.
- In the sparse format (described in [Storage Arrays for a Matrix in CSR Format \(3-Array Variation\)](#)), you store only the non-zero elements of the matrix.

In generalized eigenvalue systems you must use the same family of storage format for both matrices *A* and *B*. The bandwidth can be different for the banded format (*k1b* can be different from *k1a*), and the position of the non-zero elements can also be different for the sparse format (CSR coordinates *ib* and *jb* can be different from *ia* and *ja*).

?feast_syev/?feast_heev

Extended Eigensolver interface for standard eigenvalue problem with dense matrices.

Syntax

Fortran:

```

call sfeast_syev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res,
info)

call dfeast_syev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res,
info)

call cfeast_heev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res,
info)

call zfeast_heev(uplo, n, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m, res,
info)

```

C:

```

void sfeast_syev (const char * uplo, const MKL_INT * n, const float * a, const MKL_INT
* lda, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float
* emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float * res, MKL_INT * info);

void dfeast_syev (const char * uplo, const MKL_INT * n, const double * a, const MKL_INT
* lda, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const
double * emax, MKL_INT * m0, double * e, double * x, MKL_INT * m, double * res,
MKL_INT * info);

void cfeast_heev (const char * uplo, const MKL_INT * n, const MKL_Complex8 * a, const
MKL_INT * lda, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin,
const float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x, MKL_INT * m, float *
res, MKL_INT * info);

void zfeast_heev (const char * uplo, const MKL_INT * n, const MKL_Complex16 * a, const
MKL_INT * lda, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin,
const double * emax, MKL_INT * m0, double * e, MKL_Complex16 * x, MKL_INT * m, double
* res, MKL_INT * info);

```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

<i>uplo</i>	CHARACTER*1
	Must be 'U' or 'L' or 'F' .
	If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular parts of <i>A</i> .
	If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular parts of <i>A</i> .
	If <i>uplo</i> = 'F' , <i>a</i> stores the full matrix <i>A</i> .
<i>n</i>	INTEGER
	Sets the size of the problem. $n \geq 0$.
<i>a</i>	REAL for sfeast_syev
	DOUBLE PRECISION for dfeast_syev

COMPLEX for `cfeast_heev`

COMPLEX*16 for `zfeast_heev`

Array of dimension `lda` by `n`, contains either full matrix A or upper or lower triangular part of the matrix A , as specified by `uplo`

`lda` INTEGER

The first dimension of the array `a`. Must be at least $\max(1, n)$.

`fpm` INTEGER

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See [Extended Eigensolver Input Parameters](#) for a complete description of the parameters and their default values.

`emin, emax` REAL for `sfeast_syev` and `cfeast_heev`

DOUBLE PRECISION for `dfeast_syev` and `zfeast_heev`

The lower and upper bounds of the interval to be searched for eigenvalues; $\text{emin} \leq \text{emax}$.

`m0` INTEGER

On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where `m` is the total number of eigenvalues located in the interval $[\text{emin}, \text{emax}]$. If the initial guess is wrong, Extended Eigensolver routines return `info=3`.

`x` REAL for `sfeast_syev`

DOUBLE PRECISION for `dfeast_syev`

COMPLEX for `cfeast_heev`

COMPLEX*16 for `zfeast_heev`

On entry, if `fpm(5)=1`, the array `x(n, m)` contains a basis of guess subspace where `n` is the order of the input matrix.

Output Parameters

`epsout` REAL for `sfeast_syev` and `cfeast_heev`

DOUBLE PRECISION for `dfeast_syev` and `zfeast_heev`

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|\text{emin}|, |\text{emax}|)$

`loop` INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

`e` REAL for `sfeast_syev` and `cfeast_heev`

DOUBLE PRECISION for `dfeast_syev` and `zfeast_heev`

Array of length `m0`. On output, the first `m` entries of `e` are eigenvalues found in the interval.

<i>x</i>	On output, the first <i>m</i> columns of <i>x</i> contain the orthonormal eigenvectors corresponding to the computed eigenvalues <i>e</i> , with the <i>i</i> -th column of <i>x</i> holding the eigenvector associated with <i>e(i)</i> .
<i>m</i>	INTEGER The total number of eigenvalues found in the interval [<i>emin</i> , <i>emax</i>]: $0 \leq m \leq m_0$.
<i>res</i>	REAL for sfeast_syev and cfeast_heev DOUBLE PRECISION for dfeast_syev and zfeast_heev Array of length <i>m</i> . On exit, the first <i>m</i> components contain the relative residual vector: $\frac{\ Ax_i - \lambda_i x_i\ _1}{\max(E_{\min} , E_{\max}) \ x_i\ _1}$ for $i=1, 2, \dots, m$, and where <i>m</i> is the total number of eigenvalues found in the search interval.
<i>info</i>	INTEGER If <i>info</i> =0, the execution is successful. If <i>info</i> ≠ 0, see Output Eigensolver info Details .

?feast_sygv/?feast_hegv

Extended Eigensolver interface for generalized eigenvalue problem with dense matrices.

Syntax

Fortran:

```
call sfeast_sygv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)

call dfeast_sygv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)

call cfeast_hegv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)

call zfeast_hegv(uplo, n, a, lda, b, ldb, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)
```

C:

```
void sfeast_sygv (const char * uplo, const MKL_INT * n, const float * a, const MKL_INT *
* lda, const float * b, const MKL_INT * ldb, MKL_INT * fpm, float * epsout, MKL_INT *
loop, const float * emin, const float * emax, MKL_INT * m0, float * e, float * x,
MKL_INT * m, float * res, MKL_INT * info);

void dfeast_sygv (const char * uplo, const MKL_INT * n, const double * a, const MKL_INT *
* lda, const double * b, const MKL_INT * ldb, MKL_INT * fpm, double * epsout, MKL_INT *
loop, const double * emin, const double * emax, MKL_INT * m0, double * e, double *
x, MKL_INT * m, double * res, MKL_INT * info);
```

```
void cfeast_hegv (const char * uplo, const MKL_INT * n, const MKL_Complex8 * a, const
MKL_INT * lda, const MKL_Complex8 * b, const MKL_INT * ldb, MKL_INT * fpm, float *
epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float *
e, MKL_Complex8 * x, MKL_INT * m, float * res, MKL_INT * info);

void zfeast_hegv (const char * uplo, const MKL_INT * n, const MKL_Complex16 * a, const
MKL_INT * lda, const MKL_Complex16 * b, const MKL_INT * ldb, MKL_INT * fpm, double *
epsout, MKL_INT * loop, const double * emin, const double * emax, MKL_INT * m0, double *
e, MKL_Complex16 * x, MKL_INT * m, double * res, MKL_INT * info);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

Input Parameters

<i>uplo</i>	CHARACTER*1
	Must be 'U' or 'L' or 'F' .
	If <i>UPLO</i> = 'U', <i>a</i> and <i>b</i> store the upper triangular parts of <i>A</i> and <i>B</i> respectively.
	If <i>UPLO</i> = 'L', <i>a</i> and <i>b</i> store the lower triangular parts of <i>A</i> and <i>B</i> respectively.
	If <i>UPLO</i> = 'F', <i>a</i> and <i>b</i> store the full matrices <i>A</i> and <i>B</i> respectively.
<i>n</i>	INTEGER
	Sets the size of the problem. $n \geq 0$.
<i>a</i>	REAL for sfeast_sygv DOUBLE PRECISION for dfeast_sygv COMPLEX for cfeast_hegv COMPLEX*16 for zfeast_hegv
	Array of dimension <i>lda</i> by <i>n</i> , contains either full matrix <i>A</i> or upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i>
<i>lda</i>	INTEGER
	The first dimension of the array <i>a</i> . Must be at least max(1, <i>n</i>).
<i>b</i>	REAL for sfeast_sygv DOUBLE PRECISION for dfeast_sygv COMPLEX for cfeast_hegv COMPLEX*16 for zfeast_hegv
	Array of dimension <i>ldb</i> by <i>n</i> , contains either full matrix <i>B</i> or upper or lower triangular part of the matrix <i>B</i> , as specified by <i>uplo</i>

<i>ldb</i>	INTEGER, the first dimension of the array <i>B</i> . Must be at least $\max(1, n)$.
<i>fpm</i>	INTEGER Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<i>emin, emax</i>	REAL for <code>sfeast_sygv</code> and <code>cfeast_hegv</code> DOUBLE PRECISION for <code>dfeast_sygv</code> and <code>zfeast_hegv</code> The lower and upper bounds of the interval to be searched for eigenvalues; $\text{emin} \leq \text{emax}$.
<i>m0</i>	INTEGER On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[\text{emin}, \text{emax}]$. If the initial guess is wrong, Extended Eigensolver routines return <i>info</i> =3.
<i>x</i>	REAL for <code>sfeast_sygv</code> DOUBLE PRECISION for <code>dfeast_sygv</code> COMPLEX for <code>cfeast_hegv</code> COMPLEX*16 for <code>zfeast_hegv</code> On entry, if $fpm(5)=1$, the array $x(n, m)$ contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

<i>epsout</i>	REAL for <code>sfeast_sygv</code> and <code>cfeast_hegv</code> DOUBLE PRECISION for <code>dfeast_sygv</code> and <code>zfeast_hegv</code> On output, contains the relative error on the trace: $ trace_i - trace_{i-1} / \max(\text{emin} , \text{emax})$
<i>loop</i>	INTEGER On output, contains the number of refinement loop executed. Ignored on input.
<i>e</i>	REAL for <code>sfeast_sygv</code> and <code>cfeast_hegv</code> DOUBLE PRECISION for <code>dfeast_sygv</code> and <code>zfeast_hegv</code> Array of length $m0$. On output, the first m entries of <i>e</i> are eigenvalues found in the interval.
<i>x</i>	On output, the first m columns of <i>x</i> contain the orthonormal eigenvectors corresponding to the computed eigenvalues <i>e</i> , with the i -th column of <i>x</i> holding the eigenvector associated with <i>e</i> (<i>i</i>).
<i>m</i>	INTEGER

The total number of eigenvalues found in the interval [emin , emax]: $0 \leq m \leq m_0$.

res

REAL for `sfeast_sygv` and `cfeast_hegv`

DOUBLE PRECISION for `dfeast_sygv` and `zfeast_hegv`

Array of length m_0 . On exit, the first m components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|Bx_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info

INTEGER

If $\text{info}=0$, the execution is successful. If $\text{info} \neq 0$, see [Output Eigensolver info Details](#).

?feast_sbev/?feast_hbev

Extended Eigensolver interface for standard eigenvalue problem with banded matrices.

Syntax

Fortran:

```
call sfeast_sbev(uplo, n, kla, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)
call dfeast_sbev(uplo, n, kla, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)
call cfeast_hbev(uplo, n, kla, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)
call zfeast_hbev(uplo, n, kla, a, lda, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)
```

C:

```
void sfeast_sbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
float * a, const MKL_INT * lda, MKL_INT * fpm, float * epsout, MKL_INT * loop, const
float * emin, const float * emax, MKL_INT * m0, float * e, float * x, MKL_INT * m,
float * res, MKL_INT * info);
void dfeast_sbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
double * a, const MKL_INT * lda, MKL_INT * fpm, double * epsout, MKL_INT * loop, const
double * emin, const double * emax, MKL_INT * m0, double * e, double * x, MKL_INT * m,
double * res, MKL_INT * info);
void cfeast_hbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
MKL_Complex8 * a, const MKL_INT * lda, MKL_INT * fpm, float * epsout, MKL_INT * loop,
const float * emin, const float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x,
MKL_INT * m, float * res, MKL_INT * info);
```

```
void zfeast_hbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
MKL_Complex16 * a, const MKL_INT * lda, MKL_INT * fpm, double * epsout, MKL_INT * 
loop, const double * emin, const double * emax, MKL_INT * m0, double * e,
MKL_Complex16 * x, MKL_INT * m, double * res, MKL_INT * info);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

<i>uplo</i>	CHARACTER*1
	Must be 'U' or 'L' or 'F' .
	If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular parts of <i>A</i> .
	If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular parts of <i>A</i> .
	If <i>uplo</i> = 'F' , <i>a</i> stores the full matrix <i>A</i> .
<i>n</i>	INTEGER
	Sets the size of the problem. <i>n</i> ≥ 0 .
<i>kla</i>	INTEGER, the number of super- or sub-diagonals within the band in <i>A</i> (<i>kla</i> ≥ 0).
<i>a</i>	REAL for sfeast_sbev DOUBLE PRECISION for dfeast_sbev COMPLEX for cfeast_hbev COMPLEX*16 for zfeast_hbev Array of dimension <i>lda</i> by <i>n</i> , contains either full matrix <i>A</i> or upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i>
<i>lda</i>	INTEGER The first dimension of the array <i>a</i> . Must be at least max(1, <i>n</i>).
<i>fpm</i>	INTEGER Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<i>emin</i> , <i>emax</i>	REAL for sfeast_sbev and cfeast_hbev DOUBLE PRECISION for dfeast_sbev and zfeast_hbev The lower and upper bounds of the interval to be searched for eigenvalues; <i>emin</i> \leq <i>emax</i> .
<i>m0</i>	INTEGER

On entry, specifies the initial guess for subspace dimension to be used, $0 < m_0 \leq n$. Set $m_0 \geq m$ where m is the total number of eigenvalues located in the interval $[e_{\min}, e_{\max}]$. If the initial guess is wrong, Extended Eigensolver routines return $info=3$.

x REAL for sfeast_sbev

DOUBLE PRECISION for dfeast_sbev

COMPLEX for cfeast_hbev

COMPLEX*16 for zfeast_hbev

On entry, if $fpm(5)=1$, the array $x(n, m)$ contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

epsout

REAL for sfeast_sbev and cfeast_hbev

DOUBLE PRECISION for dfeast_sbev and zfeast_hbev

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|e_{\min}|, |e_{\max}|)$

loop

INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

e

REAL for sfeast_sbev and cfeast_hbev

DOUBLE PRECISION for dfeast_sbev and zfeast_hbev

Array of length m_0 . On output, the first m entries of *e* are eigenvalues found in the interval.

x

On output, the first m columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the i -th column of *x* holding the eigenvector associated with *e(i)*.

m

INTEGER

The total number of eigenvalues found in the interval $[e_{\min}, e_{\max}]$: $0 \leq m \leq m_0$.

res

REAL for sfeast_sbev and cfeast_hbev

DOUBLE PRECISION for dfeast_sbev and zfeast_hbev

Array of length m_0 . On exit, the first m components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i x_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|x_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info

INTEGER

If $info=0$, the execution is successful. If $info \neq 0$, see [Output Eigensolver info Details](#).

?feast_sbgv/?feast_hbgv

Extended Eigensolver interface for generalized eigenvalue problem with banded matrices.

Syntax

Fortran:

```
call sfeast_sbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)

call dfeast_sbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)

call cfeast_hbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)

call zfeast_hbgv(uplo, n, kla, a, lda, klb, b, ldb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)
```

C:

```
void sfeast_sbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
float * a, const MKL_INT * lda, const MKL_INT * klb, const float * b, const MKL_INT *
ldb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float *
emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float * res, MKL_INT * info);

void dfeast_sbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
double * a, const MKL_INT * lda, const MKL_INT * klb, const double * b, const MKL_INT *
ldb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double
* emax, MKL_INT * m0, double * e, double * x, MKL_INT * m, double * res, MKL_INT *
info);

void cfeast_hbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
MKL_Complex8 * a, const MKL_INT * lda, const MKL_INT * klb, const MKL_Complex8 * b,
const MKL_INT * ldb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float *
emin, const float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x, MKL_INT * m,
float * res, MKL_INT * info);

void zfeast_hbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const
MKL_Complex16 * a, const MKL_INT * lda, const MKL_INT * klb, const MKL_Complex16 * b,
const MKL_INT * ldb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double *
emin, const double * emax, MKL_INT * m0, double * e, MKL_Complex16 * x, MKL_INT * m,
double * res, MKL_INT * info);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

NOTE

Both matrices A and B must use the same family of storage format. The bandwidth, however, can be different (k_{lb} can be different from k_{la}).

Input Parameters

<i>uplo</i>	CHARACTER*1	
		Must be 'U' or 'L' or 'F' .
		If $UPLO = 'U'$, a and b store the upper triangular parts of A and B respectively.
		If $UPLO = 'L'$, a and b store the lower triangular parts of A and B respectively.
		If $UPLO= 'F'$, a and b store the full matrices A and B respectively.
<i>n</i>	INTEGER	
		Sets the size of the problem. $n \geq 0$.
<i>kla</i>	INTEGER	
		The number of super- or sub-diagonals within the band in A ($k_{la} \geq 0$).
<i>a</i>	REAL for sfeast_sbgv DOUBLE PRECISION for dfeast_sbgv COMPLEX for cfeast_hbgv COMPLEX*16 for zfeast_hbgv	
		Array of dimension lda by n , contains either full matrix A or upper or lower triangular part of the matrix A , as specified by <i>uplo</i>
<i>lda</i>	INTEGER	
		The first dimension of the array a . Must be at least $\max(1, n)$.
<i>klb</i>	INTEGER	
		The number of super- or sub-diagonals within the band in B ($k_{lb} \geq 0$).
<i>b</i>	REAL for sfeast_sbgv DOUBLE PRECISION for dfeast_sbgv COMPLEX for cfeast_hbgv COMPLEX*16 for zfeast_hbgv	
		Array of dimension ldb by n , contains either full matrix B or upper or lower triangular part of the matrix B , as specified by <i>uplo</i>
<i>ldb</i>	INTEGER, the first dimension of the array B . Must be at least $\max(1, n)$.	
<i>fpm</i>	INTEGER	

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See [Extended Eigensolver Input Parameters](#) for a complete description of the parameters and their default values.

emin, emax

REAL for sfeast_sbgv and cfeast_hbgv

DOUBLE PRECISION for dfeast_sbgv and zfeast_hbgv

The lower and upper bounds of the interval to be searched for eigenvalues; $\text{emin} \leq \text{emax}$.

m0

INTEGER

On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[\text{emin}, \text{emax}]$. If the initial guess is wrong, Extended Eigensolver routines return $\text{info}=3$.

x

REAL for sfeast_sbgv

DOUBLE PRECISION for dfeast_sbgv

COMPLEX for cfeast_hbgv

COMPLEX*16 for zfeast_hbgv

On entry, if $fpm(5)=1$, the array $x(n, m)$ contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

epsout

REAL for sfeast_sbgv and cfeast_hbgv

DOUBLE PRECISION for dfeast_sbgv and zfeast_hbgv

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|\text{emin}|, |\text{emax}|)$

loop

INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

e

REAL for sfeast_sbgv and cfeast_hbgv

DOUBLE PRECISION for dfeast_sbgv and zfeast_hbgv

Array of length $m0$. On output, the first m entries of *e* are eigenvalues found in the interval.

x

On output, the first m columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the i -th column of *x* holding the eigenvector associated with *e(i)*.

m

INTEGER

The total number of eigenvalues found in the interval $[\text{emin}, \text{emax}]$: $0 \leq m \leq m0$.

res

REAL for sfeast_sbgv and cfeast_hbgv

DOUBLE PRECISION for dfeast_sbgv and zfeast_hbgv

Array of length m_0 . On exit, the first m components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|Bx_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info INTEGER

If $info=0$, the execution is successful. If $info \neq 0$, see [Output Eigensolver info Details](#).

?feast_scsrev/?feast_hcsrev

Extended Eigensolver interface for standard eigenvalue problem with sparse matrices.

Syntax

Fortran:

```
call sfeast_scsrev(uplo, n, a, ia, ja, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)

call dfeast_scsrev(uplo, n, a, ia, ja, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)

call cfeast_hcsrev(uplo, n, a, ia, ja, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)

call zfeast_hcsrev(uplo, n, a, ia, ja, fpm, epsout, loop, emin, emax, m0, e, x, m,
res, info)
```

C:

```
void sfeast_scsrev (const char * uplo, const MKL_INT * n, const float * a, const
MKL_INT * ia, const MKL_INT * ja, MKL_INT * fpm, float * epsout, MKL_INT * loop, const
float * emin, const float * emax, MKL_INT * m0, float * e, float * x, MKL_INT * m,
float * res, MKL_INT * info);

void dfeast_scsrev (const char * uplo, const MKL_INT * n, const double * a, const
MKL_INT * ia, const MKL_INT * ja, MKL_INT * fpm, double * epsout, MKL_INT * loop,
const double * emin, const double * emax, MKL_INT * m0, double * e, double * x,
MKL_INT * m, double * res, MKL_INT * info);

void cfeast_hcsrev (const char * uplo, const MKL_INT * n, const MKL_Complex8 * a, const
MKL_INT * ia, const MKL_INT * ja, MKL_INT * fpm, float * epsout, MKL_INT * loop, const
float * emin, const float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x, MKL_INT * m,
float * res, MKL_INT * info);

void zfeast_hcsrev (const char * uplo, const MKL_INT * n, const MKL_Complex16 * a,
const MKL_INT * ia, const MKL_INT * ja, MKL_INT * fpm, double * epsout, MKL_INT * loop,
const double * emin, const double * emax, MKL_INT * m0, double * e,
MKL_Complex16 * x, MKL_INT * m, double * res, MKL_INT * info);
```

Include Files

- Fortran: mkl.fi

- C: mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, $Ax = \lambda x$, within a given search interval.

Input Parameters

<i>uplo</i>	CHARACTER*1	
		Must be 'U' or 'L' or 'F' .
		If <i>uplo</i> = 'U', <i>a</i> stores the upper triangular parts of <i>A</i> .
		If <i>uplo</i> = 'L', <i>a</i> stores the lower triangular parts of <i>A</i> .
		If <i>uplo</i> = 'F' , <i>a</i> stores the full matrix <i>A</i> .
<i>n</i>	INTEGER	
		Sets the size of the problem. <i>n</i> ≥ 0 .
<i>a</i>	REAL for sfeast_scsrev DOUBLE PRECISION for dfeast_scsrev COMPLEX for cfeast_hcsrev COMPLEX*16 for zfeast_hcsrev	Array containing the nonzero elements of either the full matrix <i>A</i> or the upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i> .
<i>ia</i>	INTEGER	Array of length <i>n</i> + 1, containing indices of elements in the array <i>a</i> , such that <i>ia</i> (<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> (<i>n</i> + 1) is equal to the number of non-zeros plus one.
<i>ja</i>	INTEGER	Array containing the column indices for each non-zero element of the matrix <i>A</i> being represented in the array <i>a</i> . Its length is equal to the length of the array <i>a</i> .
<i>fpm</i>	INTEGER	Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<i>emin</i> , <i>emax</i>	REAL for sfeast_scsrev and cfeast_hcsrev DOUBLE PRECISION for dfeast_scsrev and zfeast_hcsrev	The lower and upper bounds of the interval to be searched for eigenvalues; <i>emin</i> \leq <i>emax</i> .
<i>m0</i>	INTEGER	

On entry, specifies the initial guess for subspace dimension to be used, $0 < m_0 \leq n$. Set $m_0 \geq m$ where m is the total number of eigenvalues located in the interval $[e_{\min}, e_{\max}]$. If the initial guess is wrong, Extended Eigensolver routines return $\text{info}=3$.

x

```
REAL for sfeast_scsrev
DOUBLE PRECISION for dfeast_scsrev
COMPLEX for cfeast_hcsrev
COMPLEX*16 for zfeast_hcsrev
```

On entry, if $fpm(5)=1$, the array $x(n, m)$ contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

fpm

On output, the last 64 values correspond to Intel MKL PARDISO $iparm(1)$ to $iparm(64)$ (regardless of the value of $fpm(64)$ on input).

epsout

```
REAL for sfeast_scsrev and cfeast_hcsrev
DOUBLE PRECISION for dfeast_scsrev and zfeast_hcsrev
```

On output, contains the relative error on the trace: $|trace_i - trace_{i-1}| / \max(|e_{\min}|, |e_{\max}|)$

loop

INTEGER

On output, contains the number of refinement loop executed. Ignored on input.

e

```
REAL for sfeast_scsrev and cfeast_hcsrev
DOUBLE PRECISION for dfeast_scsrev and zfeast_hcsrev
```

Array of length m_0 . On output, the first m entries of *e* are eigenvalues found in the interval.

x

On output, the first m columns of *x* contain the orthonormal eigenvectors corresponding to the computed eigenvalues *e*, with the i -th column of *x* holding the eigenvector associated with *e*(i).

m

INTEGER

The total number of eigenvalues found in the interval $[e_{\min}, e_{\max}]$: $0 \leq m \leq m_0$.

res

```
REAL for sfeast_scsrev and cfeast_hcsrev
DOUBLE PRECISION for dfeast_scsrev and zfeast_hcsrev
```

Array of length m_0 . On exit, the first m components contain the relative residual vector:

$$\frac{\|Ax_i - \lambda_i x_i\|_1}{\max(|E_{\min}|, |E_{\max}|) \|x_i\|_1}$$

for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.

info INTEGER
 If $info=0$, the execution is successful. If $info \neq 0$, see [Output Eigensolver info Details](#).

?feast_scsrgv/?feast_hcsrgv

Extended Eigensolver interface for generalized eigenvalue problem with sparse matrices.

Syntax

Fortran:

```
call sfeast_scsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)

call dfeast_scsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)

call cfeast_hcsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)

call zfeast_hcsrgv(uplo, n, a, ia, ja, b, ib, jb, fpm, epsout, loop, emin, emax, m0,
e, x, m, res, info)
```

C:

```
void sfeast_scsrgv (const char * uplo, const MKL_INT * n, const float * a, const
MKL_INT * ia, const MKL_INT * ja, const float * b, const MKL_INT * ib, const MKL_INT *
jb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float *
emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float * res, MKL_INT * info);

void dfeast_scsrgv (const char * uplo, const MKL_INT * n, const double * a, const
MKL_INT * ia, const MKL_INT * ja, const double * b, const MKL_INT * ib, const MKL_INT *
jb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double
* emax, MKL_INT * m0, double * e, double * x, MKL_INT * m, double * res, MKL_INT *
info);

void cfeast_hcsrgv (const char * uplo, const MKL_INT * n, const MKL_Complex8 * a, const
MKL_INT * ia, const MKL_INT * ja, const MKL_Complex8 * b, const MKL_INT * ib, const
MKL_INT * jb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const
float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x, MKL_INT * m, float * res,
MKL_INT * info);

void zfeast_hcsrgv (const char * uplo, const MKL_INT * n, const MKL_Complex16 * a,
const MKL_INT * ia, const MKL_INT * ja, const MKL_Complex16 * b, const MKL_INT * ib,
const MKL_INT * jb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double *
emin, const double * emax, MKL_INT * m0, double * e, MKL_Complex16 * x, MKL_INT * m,
double * res, MKL_INT * info);
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, $Ax = \lambda Bx$, within a given search interval.

NOTE

Both matrices A and B must use the same family of storage format. The position of the non-zero elements can be different (CSR coordinates ib and jb can be different from ia and ja).

Input Parameters

<i>uplo</i>	CHARACTER*1	
	Must be 'U' or 'L' or 'F' .	
	If <i>UPLO</i> = 'U', <i>a</i> and <i>b</i> store the upper triangular parts of <i>A</i> and <i>B</i> respectively.	
	If <i>UPLO</i> = 'L', <i>a</i> and <i>b</i> store the lower triangular parts of <i>A</i> and <i>B</i> respectively.	
	If <i>UPLO</i> = 'F', <i>a</i> and <i>b</i> store the full matrices <i>A</i> and <i>B</i> respectively.	
<i>n</i>	INTEGER	
	Sets the size of the problem. $n \geq 0$.	
<i>a</i>	REAL for sfeast_scsrgv DOUBLE PRECISION for dfeast_scsrgv COMPLEX for cfeast_hcsrgv COMPLEX*16 for zfeast_hcsrgv	
	Array containing the nonzero elements of either the full matrix <i>A</i> or the upper or lower triangular part of the matrix <i>A</i> , as specified by <i>uplo</i> .	
<i>ia</i>	INTEGER	
	Array of length $n + 1$, containing indices of elements in the array <i>a</i> , such that <i>ia</i> (<i>i</i>) is the index in the array <i>a</i> of the first non-zero element from the row <i>i</i> . The value of the last element <i>ia</i> ($n + 1$) is equal to the number of non-zeros plus one.	
<i>ja</i>	INTEGER	
	Array containing the column indices for each non-zero element of the matrix <i>A</i> being represented in the array <i>a</i> . Its length is equal to the length of the array <i>a</i> .	
<i>b</i>	REAL for sfeast_scsrgv DOUBLE PRECISION for dfeast_scsrgv COMPLEX for cfeast_hcsrgv COMPLEX*16 for zfeast_hcsrgv	
	Array of dimension <i>ldb</i> by *, contains either full matrix <i>B</i> or upper or lower triangular part of the matrix <i>B</i> , as specified by <i>uplo</i>	

<i>ib</i>	INTEGER	Array of length $n + 1$, containing indices of elements in the array b , such that $ib(i)$ is the index in the array b of the first non-zero element from the row i . The value of the last element $ib(n + 1)$ is equal to the number of non-zeros plus one.
<i>jb</i>	INTEGER	Array containing the column indices for each non-zero element of the matrix B being represented in the array b . Its length is equal to the length of the array b .
<i>fpm</i>	INTEGER	Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
<i>emin</i> , <i>emax</i>	REAL for <code>sfeast_scsrgv</code> and <code>cfeast_hcsrgv</code> DOUBLE PRECISION for <code>dfeast_scsrgv</code> and <code>zfeast_hcsrgv</code>	The lower and upper bounds of the interval to be searched for eigenvalues; $emin \leq emax$.
<i>m0</i>	INTEGER	On entry, specifies the initial guess for subspace dimension to be used, $0 < m0 \leq n$. Set $m0 \geq m$ where m is the total number of eigenvalues located in the interval $[emin, emax]$. If the initial guess is wrong, Extended Eigensolver routines return <code>info=3</code> .
<i>x</i>	REAL for <code>sfeast_scsrgv</code> DOUBLE PRECISION for <code>dfeast_scsrgv</code> COMPLEX for <code>cfeast_hcsrgv</code> COMPLEX*16 for <code>zfeast_hcsrgv</code>	On entry, if $fpm(5)=1$, the array $x(n, m)$ contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

<i>fpm</i>	On output, the last 64 values correspond to Intel MKL PARDISO <code>iparm(1)</code> to <code>iparm(64)</code> (regardless of the value of <code>fpm(64)</code> on input).
<i>epsout</i>	REAL for <code>sfeast_scsrgv</code> and <code>cfeast_hcsrgv</code> DOUBLE PRECISION for <code>dfeast_scsrgv</code> and <code>zfeast_hcsrgv</code> On output, contains the relative error on the trace: $ trace_i - trace_{i-1} / \max(emin , emax)$
<i>loop</i>	INTEGER On output, contains the number of refinement loop executed. Ignored on input.

<i>e</i>	REAL for sfeast_scsrgv and cfeast_hcsrgv DOUBLE PRECISION for dfeast_scsrgv and zfeast_hcsrgv Array of length m_0 . On output, the first m entries of <i>e</i> are eigenvalues found in the interval.
<i>x</i>	On output, the first m columns of <i>x</i> contain the orthonormal eigenvectors corresponding to the computed eigenvalues <i>e</i> , with the i -th column of <i>x</i> holding the eigenvector associated with <i>e</i> (i).
<i>m</i>	INTEGER The total number of eigenvalues found in the interval [e_{\min} , e_{\max}]: $0 \leq m \leq m_0$.
<i>res</i>	REAL for sfeast_scsrgv and cfeast_hcsrgv DOUBLE PRECISION for dfeast_scsrgv and zfeast_hcsrgv Array of length m_0 . On exit, the first m components contain the relative residual vector:
	$\frac{\ Ax_i - \lambda_i Bx_i\ _1}{\max(E_{\min} , E_{\max}) \ Bx_i\ _1}$ for $i=1, 2, \dots, m$, and where m is the total number of eigenvalues found in the search interval.
<i>info</i>	INTEGER If <i>info</i> =0, the execution is successful. If <i>info</i> ≠ 0, see Output Eigensolver info Details .

Vector Mathematical Functions

This chapter describes Intel® MKL Vector Mathematical Functions Library (VML), which computes a mathematical function of each of the vector elements. VML includes a set of highly optimized functions (arithmetic, power, trigonometric, exponential, hyperbolic, special, and rounding) that operate on vectors of real and complex numbers.

Application programs that improve performance with VML include nonlinear programming software, computation of integrals, financial calculations, computer graphics, and many others.

VML functions fall into the following groups according to the operations they perform:

- [VML Mathematical Functions](#) compute values of mathematical functions, such as sine, cosine, exponential, or logarithm, on vectors stored contiguously in memory.
- [VML Pack/Unpack Functions](#) convert to and from vectors with positive increment indexing, vector indexing, and mask indexing (see [Appendix B](#) for details on vector indexing methods).
- [VML Service Functions](#) set/get the accuracy modes and the error codes, and free memory.

The VML mathematical functions take an input vector as an argument, compute values of the respective function element-wise, and return the results in an output vector. All the VML mathematical functions can perform in-place operations, where the input and output arrays are at the same memory locations.

The Intel MKL interfaces are given in the following include files:

- `mkl_vml.f77`, which declares the FORTRAN 77 interfaces
- `mkl_vml.f90`, which declares the Fortran 90 interfaces; the `mkl_vml.fi` include file available in the previous versions of Intel MKL is retained for backward compatibility
- `mkl_vml_functions.h`, which declares the C interfaces

Examples that demonstrate how to use the VML functions are located in:

```
 ${MKL}/examples/vmlc/source
```

```
 ${MKL}/examples/vmlf/source
```

See VML performance and accuracy data in the online VML Performance and Accuracy Data document available at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Data Types, Accuracy Modes, and Performance Tips

VML includes mathematical and pack/unpack vector functions for single and double precision vector arguments of real and complex types. The library provides Fortran- and C-interfaces for all functions, including the associated service functions. The Function Naming Conventions section below shows how to call these functions from different languages.

Performance depends on a number of factors, including vectorization and threading overhead. The recommended usage is as follows:

- Use VML for vector lengths larger than 40 elements.
 - Use the Intel® Compiler for vector lengths less than 40 elements.

All VML vector functions support the following accuracy modes:

- High Accuracy (HA), the default mode
 - Low Accuracy (LA), which improves performance by reducing accuracy of the two least significant bits
 - Enhanced Performance (EP), which provides better performance at the cost of significantly reduced accuracy. Approximately half of the bits in the mantissa are correct.

Note that using the EP mode does not guarantee accurate processing of corner cases and special values. Although the default accuracy is HA, LA is sufficient in most cases. For applications that require less accuracy (for example, media applications, some Monte Carlo simulations, etc.), the EP mode may be sufficient.

VML handles special values in accordance with the C99 standard [C99].

Intel MKL offers both functions and environment variables to switch between modes for VML. See the *Intel MKL User's Guide* for details about the environment variables. Use the `vmlSetMode(mode)` function (see [Table "Values of the mode Parameter"](#)) to switch between the HA, LA, and EP modes. The `vmlGetMode()` function returns the current mode.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

See Also

Function Naming Conventions

Function Naming Conventions

The VML function names are lowercase for Fortran (`vsabs`) and of mixed (lower and upper) case for C (`vsAbs`).

The VML mathematical and pack/unpack function names have the following structure:

v[m]<?><name><mod>

where

- v is a prefix indicating vector operations.
 - $[m]$ is an optional prefix for mathematical functions that indicates additional argument to specify a VML mode for a given function call (see [vmlSetMode](#) for possible values and their description).
 - $<?>$ is a precision prefix that indicates one of the following the data types:

s REAL (KIND=4) for the Fortran interface, or float for the C interface.

d REAL (KIND=8 for the Fortran interface, or double for the C interface).

C COMPLEX (KIND=4) for the Fortran interface, or `MKL_Complex8` for the C interface.

z COMPLEX (KIND=8) for the Fortran interface, or MKL_Complex16 for the C interface.

- <name> indicates the function short name, with some of its letters in uppercase for the C interface. See examples in [Table "VML Mathematical Functions"](#).
- <mod> field (written in uppercase for the C interface) is present only in the pack/unpack functions and indicates the indexing method used:

<i>i</i>	indexing with a positive increment
<i>v</i>	indexing with an index vector
<i>m</i>	indexing with a mask vector.

The VML service function names have the following structure:

vm/*<name>*

where

<name> indicates the function short name, with some of its letters in uppercase for the C interface. See examples in [Table "VML Service Functions"](#).

To call VML functions from an application program, use conventional function calls. For example, call the vector single precision real exponential function as

call vsexp (*n*, *a*, *y*) for the Fortran interface

call vmsexp (*n*, *a*, *y*, *mode*) for the Fortran interface with a specified mode

vsExp (*n*, *a*, *y*); for the C interface

Function Interfaces

VML interfaces include the function names and argument lists. The following sections describe the interfaces for the VML functions. Note that some of the functions have multiple input and output arguments

Some VML functions may also take scalar arguments as input. See the function description for the naming conventions of such arguments.

VML Mathematical Functions

Fortran:

```
call v<?><name>( n, a, [scalar input arguments,] y )
call v<?><name>( n, a, b, [scalar input arguments,] y )
call v<?><name>( n, a, y, z )
call vm<?><name>( n, a, [scalar input arguments,] y, mode )
call vm<?><name>( n, a, b, [scalar input arguments,] y, mode )
call vm<?><name>( n, a, y, z, mode )
```

C:

```
v<?><name>( n, a, [scalar input arguments,] y );
v<?><name>( n, a, b, [scalar input arguments,] y );
v<?><name>( n, a, y, z );
vm<?><name>( n, a, [scalar input arguments,] y, mode );
vm<?><name>( n, a, b, [scalar input arguments,] y, mode );
vm<?><name>( n, a, y, z, mode );
```

Pack Functions

Fortran:

```
call v<?>packi( n, a, inca, y )
call v<?>packv( n, a, ia, y )
call v<?>packm( n, a, ma, y )
```

C:

```
v<?>PackI( n, a, inca, y );
v<?>PackV( n, a, ia, y );
v<?>PackM( n, a, ma, y );
```

Unpack Functions

Fortran:

```
call v<?>unpacki( n, a, y, incy )
call v<?>unpackv( n, a, y, iy )
call v<?>unpackm( n, a, y, my )
```

C:

```
v<?>UnpackI( n, a, y, incy );
v<?>UnpackV( n, a, y, iy );
v<?>UnpackM( n, a, y, my );
```

Service Functions

Fortran:

```
oldmode = vmlsetmode( mode )
mode = vmlgetmode( )
olderr = vmlseterrstatus( err )
err = vmlgeterrstatus( )
olderr = vmlclearerrstatus( )
oldcallback = vmlseterrorcallback( callback )
callback = vmlgeterrorcallback( )
oldcallback = vmlclearerrorcallback( )
```

C:

```
oldmode = vmlSetMode( mode );
mode = vmlGetMode( void );
olderr = vmlSetErrStatus( err );
err = vmlGetErrStatus( void );
olderr = vmlClearErrStatus( void );
oldcallback = vmlsetErrorCallBack( callback );
callback = vmlgetErrorCallBack( void );
oldcallback = vmlClearErrorCallBack( void );
```

Note that *oldmode*, *olderr*, and *oldcallback* refer to settings prior to the call.

Input Parameters

<i>n</i>	number of elements to be calculated
<i>a</i>	first input vector
<i>b</i>	second input vector
<i>inca</i>	vector increment for the input vector <i>a</i>
<i>ia</i>	index vector for the input vector <i>a</i>
<i>ma</i>	mask vector for the input vector <i>a</i>
<i>incy</i>	vector increment for the output vector <i>y</i>
<i>iy</i>	index vector for the output vector <i>y</i>
<i>my</i>	mask vector for the output vector <i>y</i>
<i>err</i>	error code
<i>mode</i>	VML mode
<i>callback</i>	address of the callback function

Output Parameters

<i>y</i>	first output vector
<i>z</i>	second output vector
<i>err</i>	error code
<i>mode</i>	VML mode
<i>olderr</i>	former error code
<i>oldmode</i>	former VML mode
<i>callback</i>	address of the callback function
<i>oldcallback</i>	address of the former callback function

See the data types of the parameters used in each function in the respective function description section. All the Intel MKL VML mathematical functions can perform in-place operations.

Vector Indexing Methods

VML mathematical functions work only with unit stride. To accommodate arrays with other increments, or more complicated indexing, you can gather the elements into a contiguous vector and then scatter them after the computation is complete.

VML uses the three following indexing methods to do this task:

- positive increment
- index vector
- mask vector

The indexing method used in a particular function is indicated by the indexing modifier (see the description of the `<mod>` field in [Function Naming Conventions](#)). For more information on the indexing methods, see [Vector Arguments in VML](#) in Appendix B.

Error Diagnostics

The VML mathematical functions incorporate the error handling mechanism, which is controlled by the following service functions:

<code>vmlGetErrStatus,</code> <code>vmlSetErrStatus,</code> <code>vmlClearErrStatus</code>	These functions operate with a global variable called VML Error Status. The VML Error Status flags an error, a warning, or a successful execution of a VML function.
<code>vmlGetErrCallBack,</code> <code>vmlSetErrCallBack,</code> <code>vmlClearErrCallBack</code>	These functions enable you to customize the error handling. For example, you can identify a particular argument in a vector where an error occurred or that caused a warning.
<code>vmlSetMode, vmlGetMode</code>	These functions get and set a VML mode. If you set a new VML mode using the <code>vmlSetMode</code> function, you can store the previous VML mode returned by the routine and restore it at any point of your application.

If both an error and a warning situation occur during the function call, the VML Error Status variable keeps only the value of the error code. See [Table "Values of the VML Error Status"](#) for possible values. If a VML function does not encounter errors or warnings, it sets the VML Error Status to `VML_STATUS_OK`.

If you use incorrect input arguments to a VML function (`VML_STATUS_BADSIZE` and `VML_STATUS_BADMEM`), the function calls `xerbla` to report errors. See [Table "Values of the VML Error Status"](#) for details

You can use the `vmlSetMode` and `vmlGetMode` functions to modify error handling behavior. Depending on the VML mode, the error handling behavior includes the following operations:

- setting the VML Error Status to a value corresponding to the observed error or warning
- setting the `errno` variable to one of the values described in [Table "Set Values of the `errno` Variable"](#)
- writing error text information to the `stderr` stream
- raising the appropriate exception on an error, if necessary
- calling the additional error handler callback function that is set by `vmlSetErrorCallBack`.

Set Values of the `errno` Variable

Value of <code>errno</code>	Description
<code>0</code>	No errors are detected.
<code>EINVAL</code>	The array dimension is not positive.
<code>EACCES</code>	NULL pointer is passed.
<code>EDOM</code>	At least one of array values is out of a range of definition.
<code>ERANGE</code>	At least one of array values caused a singularity, overflow or underflow.

See Also

[vmlGetErrStatus](#) Gets the VML Error Status.

[vmlSetErrStatus](#) Sets the new VML Error Status according to `err` and stores the previous VML Error Status to `olderr`.

[vmlClearErrStatus](#) Sets the VML Error Status to `VML_STATUS_OK` and stores the previous VML Error Status to `olderr`.

[vmlSetErrorCallBack](#) Sets the additional error handler callback function and gets the old callback function.

[vmlGetErrorCallBack](#) Gets the additional error handler callback function.

`vmlClearColorCallBack` Deletes the additional error handler callback function and retrieves the former callback function.

`vmlGetMode` Gets the VML mode.

`vmlSetMode` Sets a new mode for VML functions according to the *mode* parameter and stores the previous VML mode to *oldmode*.

VML Mathematical Functions

This section describes VML functions that compute values of mathematical functions on real and complex vector arguments with unit increment.

Each function is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data, as well as a description of the input/output arguments.

The input range of parameters is equal to the mathematical range of the input data type, unless the function description specifies input threshold values, which mark off the precision overflow, as follows:

- `FLOAT_MAX` denotes the maximum number representable in single precision real data type
- `DBL_MAX` denotes the maximum number representable in double precision real data type

Table "VML Mathematical Functions" lists available mathematical functions and associated data types.

VML Mathematical Functions

Function	Data Types	Description
Arithmetic Functions		
<code>v?Add</code>	<i>s, d, c, z</i>	Addition of vector elements
<code>v?Sub</code>	<i>s, d, c, z</i>	Subtraction of vector elements
<code>v?Sqr</code>	<i>s, d</i>	Squaring of vector elements
<code>v?Mul</code>	<i>s, d, c, z</i>	Multiplication of vector elements
<code>v?MulByConj</code>	<i>c, z</i>	Multiplication of elements of one vector by conjugated elements of the second vector
<code>v?Conj</code>	<i>c, z</i>	Conjugation of vector elements
<code>v?Abs</code>	<i>s, d, c, z</i>	Computation of the absolute value of vector elements
<code>v?Arg</code>	<i>c, z</i>	Computation of the argument of vector elements
<code>v?LinearFrac</code>	<i>s, d</i>	Linear fraction transformation of vectors
Power and Root Functions		
<code>v?Inv</code>	<i>s, d</i>	Inversion of vector elements
<code>v?Div</code>	<i>s, d, c, z</i>	Division of elements of one vector by elements of the second vector
<code>v?Sqrt</code>	<i>s, d, c, z</i>	Computation of the square root of vector elements
<code>v?InvSqrt</code>	<i>s, d</i>	Computation of the inverse square root of vector elements
<code>v?Cbrt</code>	<i>s, d</i>	Computation of the cube root of vector elements
<code>v?InvCbrt</code>	<i>s, d</i>	Computation of the inverse cube root of vector elements
<code>v?Pow2o3</code>	<i>s, d</i>	Raising each vector element to the power of 2/3
<code>v?Pow3o2</code>	<i>s, d</i>	Raising each vector element to the power of 3/2
<code>v?Pow</code>	<i>s, d, c, z</i>	Raising each vector element to the specified power
<code>v?Powx</code>	<i>s, d, c, z</i>	Raising each vector element to the constant power
<code>v?Hypot</code>	<i>s, d</i>	Computation of the square root of sum of squares
Exponential and Logarithmic Functions		
<code>v?Exp</code>	<i>s, d, c, z</i>	Computation of the exponential of vector elements
<code>v?Expm1</code>	<i>s, d</i>	Computation of the exponential of vector elements decreased by 1
<code>v?Ln</code>	<i>s, d, c, z</i>	Computation of the natural logarithm of vector elements
<code>v?Log10</code>	<i>s, d, c, z</i>	Computation of the denary logarithm of vector elements
<code>v?Log1p</code>	<i>s, d</i>	Computation of the natural logarithm of vector elements that are increased by 1
Trigonometric Functions		
<code>v?Cos</code>	<i>s, d, c, z</i>	Computation of the cosine of vector elements

Function	Data Types	Description
v?Sin	s, d, c, z	Computation of the sine of vector elements
v?SinCos	s, d	Computation of the sine and cosine of vector elements
v?CIS	c, z	Computation of the complex exponent of vector elements (cosine and sine combined to complex value)
v?Tan	s, d, c, z	Computation of the tangent of vector elements
v?Acos	s, d, c, z	Computation of the inverse cosine of vector elements
v?Asin	s, d, c, z	Computation of the inverse sine of vector elements
v?Atan	s, d, c, z	Computation of the inverse tangent of vector elements
v?Atan2	s, d	Computation of the four-quadrant inverse tangent of elements of two vectors
Hyperbolic Functions		
v?Cosh	s, d, c, z	Computation of the hyperbolic cosine of vector elements
v?Sinh	s, d, c, z	Computation of the hyperbolic sine of vector elements
v?Tanh	s, d, c, z	Computation of the hyperbolic tangent of vector elements
v?Acosh	s, d, c, z	Computation of the inverse hyperbolic cosine of vector elements
v?Asinh	s, d, c, z	Computation of the inverse hyperbolic sine of vector elements
v?Atanh	s, d, c, z	Computation of the inverse hyperbolic tangent of vector elements.
Special Functions		
v?Erf	s, d	Computation of the error function value of vector elements
v?Erfc	s, d	Computation of the complementary error function value of vector elements
v?CdfNorm	s, d	Computation of the cumulative normal distribution function value of vector elements
v?ErfInv	s, d	Computation of the inverse error function value of vector elements
v?ErfcInv	s, d	Computation of the inverse complementary error function value of vector elements
v?CdfNormInv	s, d	Computation of the inverse cumulative normal distribution function value of vector elements
v?LGamma	s, d	Computation of the natural logarithm for the absolute value of the gamma function of vector elements
v?TGamma	s, d	Computation of the gamma function of vector elements
Rounding Functions		
v?Floor	s, d	Rounding towards minus infinity
v?Ceil	s, d	Rounding towards plus infinity
v?Trunc	s, d	Rounding towards zero infinity
v?Round	s, d	Rounding to nearest integer
v?NearbyInt	s, d	Rounding according to current mode
v?Rint	s, d	Rounding according to current mode and raising inexact result exception
v?Modf	s, d	Computation of the integer and fractional parts
v?Frac	s, d	Computation of the fractional part

Special Value Notations

This section defines notations of special values for complex functions. The definitions are provided in text, tables, or formulas.

- z, z_1, z_2 , etc. denote complex numbers.
- $i, i^2=-1$ is the imaginary unit.
- x, X, x_1, x_2 , etc. denote real imaginary parts.
- y, Y, y_1, y_2 , etc. denote imaginary parts.
- x and Y represent any finite positive IEEE-754 floating point values, if not stated otherwise.
- Quiet NaN and signaling NaN are denoted with QNAN and SNAN, respectively.
- The IEEE-754 positive infinities or floating-point numbers are denoted with a + sign before X, Y , etc.

- The IEEE-754 negative infinities or floating-point numbers are denoted with a - sign before X, Y, etc.

`CONJ(z)` and `CIS(z)` are defined as follows:

$$\text{CONJ}(x+i\cdot y) = x - i \cdot y$$

$$\text{CIS}(y) = \cos(y) + i \cdot \sin(y).$$

The special value tables show the result of the function for the `z` argument at the intersection of the `RE(z)` column and the `i*IM(z)` row. If the function raises an exception on the argument `z`, the lower part of this cell shows the raised exception and the VML Error Status. An empty cell indicates that this argument is normal and the result is defined mathematically.

Arithmetic Functions

Arithmetic functions perform the basic mathematical operations like addition, subtraction, multiplication or computation of the absolute value of the vector elements.

v?Add

Performs element by element addition of vector `a` and vector `b`.

Syntax

Fortran:

```
call vsadd( n, a, b, y )
call vmsadd( n, a, b, y, mode )
call vdadd( n, a, b, y )
call vmdadd( n, a, b, y, mode )
call vcadd( n, a, b, y )
call vmcadd( n, a, b, y, mode )
call vzadd( n, a, b, y )
call vmzadd( n, a, b, y, mode )
```

C:

```
vsAdd( n, a, b, y );
vmsAdd( n, a, b, y, mode );
vdAdd( n, a, b, y );
vmdAdd( n, a, b, y, mode );
vcAdd( n, a, b, y );
vmcAdd( n, a, b, y, mode );
vzAdd( n, a, b, y );
vmzAdd( n, a, b, y, mode );
```

Include Files

- Fortran: `mkl_vml.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	FORTRAN 77: REAL for vsAdd, vmsAdd DOUBLE PRECISION for vdadd, vmdadd COMPLEX for vcadd, vmcadd DOUBLE COMPLEX for vzadd, vmzadd Fortran 90: REAL, INTENT(IN) for vsAdd, vmsAdd DOUBLE PRECISION, INTENT(IN) for vdadd, vmdadd COMPLEX, INTENT(IN) for vcadd, vmcadd DOUBLE COMPLEX, INTENT(IN) for vzadd, vmzadd C: const float* for vsAdd, vmsAdd const double* for vdAdd, vmdAdd const MKL_Complex8* for vcAdd, vmcAdd const MKL_Complex16* for vzAdd, vmzAdd	Fortran: Arrays that specify the input vectors <i>a</i> and <i>b</i> . C: Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>mode</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vsAdd, vmsAdd DOUBLE PRECISION for vdadd, vmdadd COMPLEX, for vcadd, vmcadd DOUBLE COMPLEX for vzadd, vmzadd Fortran 90: REAL, INTENT(OUT) for vsAdd, vmsAdd	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
	DOUBLE PRECISION, INTENT (OUT) for vdadd, vmdadd	
	COMPLEX, INTENT (OUT) for vcadd, vmcadd	
	DOUBLE COMPLEX, INTENT (OUT) for vzadd, vmzadd	
C:	float* for vsAdd, vmsAdd double* for vdAdd, vmdAdd	
	MKL_Complex8* for vcAdd, vmcAdd	
	MKL_Complex16* for vzAdd, vmzAdd	

Description

The v?Add function performs element by element addition of vector *a* and vector *b*.

Special values for Real Function v?Add(x)

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	+0	
-0	+0	+0	
-0	-0	-0	
+∞	+∞	+∞	
+∞	-∞	QNAN	INVALID
-∞	+∞	QNAN	INVALID
-∞	-∞	-∞	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Add}(x_1 + i \cdot y_1, x_2 + i \cdot y_2) = (x_1 + x_2) + i \cdot (y_1 + y_2)$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when *x*₁, *x*₂, *y*₁, *y*₂ are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the OVERFLOW exception, and sets the VML Error Status to VML_STATUS_OVERFLOW (overriding any possible VML_STATUS_ACCURACYWARNING status).

v?Sub

Performs element by element subtraction of vector *b* from vector *a*.

Syntax

Fortran:

```
call vssub( n, a, b, y )
call vmssub( n, a, b, y, mode )
call vdsub( n, a, b, y )
```

```
call vmdsub( n, a, b, y, mode )
call vcsub( n, a, b, y )
call vmcsub( n, a, b, y, mode )
call vzsub( n, a, b, y )
call vmzsub( n, a, b, y, mode )
```

C:

```
vsSub( n, a, b, y );
vmsSub( n, a, b, y, mode );
vdSub( n, a, b, y );
vmdSub( n, a, b, y, mode );
vcSub( n, a, b, y );
vmcSub( n, a, b, y, mode );
vzSub( n, a, b, y );
vmzSub( n, a, b, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER, INTENT(IN) <small>C:</small> const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	<small>FORTRAN 77:</small> REAL for vssub, vmssub <small>DOUBLE PRECISION</small> for vdsub, vmdsub <small>COMPLEX</small> for vcsub, vmcsub <small>DOUBLE COMPLEX</small> for vzsub, vmzsub <small>Fortran 90:</small> REAL, INTENT(IN) for vssub, vmssub <small>DOUBLE PRECISION, INTENT(IN)</small> for vdsub, vmdsub <small>COMPLEX, INTENT(IN)</small> for vcsub, vmcsub <small>DOUBLE COMPLEX, INTENT(IN)</small> for vzsub, vmzsub <small>C:</small> const float* for vsSub, vmsSub const double* for vdSub, vmdSub	<small>Fortran:</small> Arrays that specify the input vectors <i>a</i> and <i>b</i> . <small>C:</small> Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .

Name	Type	Description
	const MKL_Complex8* for vcSub, vmcSub	
	const MKL_Complex16* for vzSub, vmzSub	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vssub, vmssub DOUBLE PRECISION for vdsub, vmdsub COMPLEX for vcsub, vmcsu DOUBLE COMPLEX for vzsub, vmzsub Fortran 90: REALINTENT(OUT) for vssub, vmssub DOUBLE PRECISION, INTENT(OUT) for vdsub, vmdsub COMPLEX, INTENT(OUT) for vcsub, vmcsu DOUBLE COMPLEX, INTENT(OUT) for vzsub, vmzsub C: float* for vsSub, vmsSub double* for vdSub, vmdSub MKL_Complex8* for vcSub, vmcSub MKL_Complex16* for vzSub, vmzSub	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The v?Sub function performs element by element subtraction of vector *b* from vector *a*.

Special values for Real Function v?Sub(x)

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	+0	
-0	+0	-0	
-0	-0	+0	
+∞	+∞	QNAN	INVALID
+∞	-∞	+∞	
-∞	+∞	-∞	

Argument 1	Argument 2	Result	Exception
-∞	-∞	QNAN	INVALID
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Sub}(x_1+i*y_1, x_2+i*y_2) = (x_1-x_2) + i*(y_1-y_2).$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when x_1, x_2, y_1, y_2 are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the OVERFLOW exception, and sets the VML Error Status to VML_STATUS_OVERFLOW (overriding any possible VML_STATUS_ACCURACYWARNING status).

v?Sqr

Performs element by element squaring of the vector.

Syntax

Fortran:

```
call vssqr( n, a, y )
call vmssqr( n, a, y, mode )
call vdsqr( n, a, y )
call vmdsqr( n, a, y, mode )
```

C:

```
vsSqr( n, a, y );
vmsSqr( n, a, y, mode );
vdSqr( n, a, y );
vmdSqr( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTRAN 77 : INTEGER Fortran 90 : INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
a	FORTRAN 77 : REAL for vssqr, vmssqr	Fortran : Array that specifies the input vector <i>a</i> . C : Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	DOUBLE PRECISION for vdsqr, vmdsqr Fortran 90: REAL, INTENT(IN) for vssqr, vmssqr	
	DOUBLE PRECISION, INTENT(IN) for vdsqr, vmdsqr	
	C: const float* for vsSqr, vmsSqr const double* for vdSqr, vmdSqr	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vssqr, vmssqr DOUBLE PRECISION for vdsqr, vmdsqr Fortran 90: REAL, INTENT(OUT) for vssqr, vmssqr DOUBLE PRECISION, INTENT(OUT) for vdsqr, vmdsqr C: float* for vsSqr, vmsSqr double* for vdSqr, vmdSqr	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The `v?Sqr` function performs element by element squaring of the vector.

Special Values for Real Function `v?Sqr(x)`

Argument	Result	Exception
+0	+0	
-0	+0	
+∞	+∞	
-∞	+∞	
QNAN	QNAN	
SNAN	QNAN	INVALID

v?Mul

Performs element by element multiplication of vector *a* and vector *b*.

Syntax

Fortran:

```
call vsmul( n, a, b, y )
call vmsmul( n, a, b, y, mode )
call vdmul( n, a, b, y )
call vmdmul( n, a, b, y, mode )
call vcmul( n, a, b, y )
call vmcmul( n, a, b, y, mode )
call vzmul( n, a, b, y )
call vmzmul( n, a, b, y, mode )
```

C:

```
vsMul( n, a, b, y );
vmsMul( n, a, b, y, mode );
vdMul( n, a, b, y );
vmdMul( n, a, b, y, mode );
vcMul( n, a, b, y );
vmcMul( n, a, b, y, mode );
vzMul( n, a, b, y );
vmzMul( n, a, b, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	FORTRAN 77: REAL for vsmul, vmsmul DOUBLE PRECISION for vdmul, vmdmul COMPLEX for vcmul, vmcmul DOUBLE COMPLEX for vzmul, vmzmul Fortran 90: REAL, INTENT(IN) for vsmul, vmsmul DOUBLE PRECISION, INTENT(IN) for vdmul, vmdmul	Fortran: Arrays that specify the input vectors <i>a</i> and <i>b</i> . C: Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .

Name	Type	Description
	COMPLEX, INTENT(IN) for vcmul, vmcmul DOUBLE COMPLEX, INTENT(IN) for vzmul, vmzmul C: const float* for vsMul, vmsMul const double* for vdMul, vmdMul const MKL_Complex8* for vcMul, vmcMul const MKL_Complex16* for vzMul, vmzMul	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vsmul, vmsmul DOUBLE PRECISION for vdmul, vmdmul COMPLEX, for vcmul, vmcmul DOUBLE COMPLEX for vzmul, vmzmul Fortran 90: REAL, INTENT(OUT) for vsmul, vmsmul DOUBLE PRECISION, INTENT(OUT) for vdmul, vmdmul COMPLEX, INTENT(OUT) for vcmul, vmcmul DOUBLE COMPLEX, INTENT(OUT) for vzmul, vmzmul C: float* for vsMul, vmsMul double* for vdMul, vmdMul MKL_Complex8* for vcMul, vmcMul MKL_Complex16* for vzMul, vmzMul	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Mul` function performs element by element multiplication of vector *a* and vector *b*.

Special values for Real Function v?Mul(x)

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
+0	-0	-0	
-0	+0	-0	
-0	-0	+0	
+0	+∞	QNAN	INVALID
+0	-∞	QNAN	INVALID
-0	+∞	QNAN	INVALID
-0	-∞	QNAN	INVALID
+∞	+0	QNAN	INVALID
+∞	-0	QNAN	INVALID
-∞	+0	QNAN	INVALID
-∞	-0	QNAN	INVALID
+∞	+∞	+∞	
+∞	-∞	-∞	
-∞	+∞	-∞	
-∞	-∞	+∞	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	non-SNAN	QNAN	
non-SNAN	QNAN	QNAN	

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Mul}(x_1 + i \cdot y_1, x_2 + i \cdot y_2) = (x_1 \cdot x_2 - y_1 \cdot y_2) + i \cdot (x_1 \cdot y_2 + y_1 \cdot x_2).$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when x_1, x_2, y_1, y_2 are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the OVERFLOW exception, and sets the VML Error Status to VML_STATUS_OVERFLOW (overriding any possible VML_STATUS_ACCURACYWARNING status).

v?MulByConj

Performs element by element multiplication of vector *a* element and conjugated vector *b* element.

Syntax**Fortran:**

```
call vcmulbyconj( n, a, b, y )
call vmcmulbyconj( n, a, b, y, mode )
call vzmulbyconj( n, a, b, y )
call vmzmulbyconj( n, a, b, y, mode )
```

C:

```
vcMulByConj( n, a, b, y );
vmcMulByConj( n, a, b, y, mode );
vzMulByConj( n, a, b, y );
vmzMulByConj( n, a, b, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER, INTENT(IN)</p> <p>C: const MKL_INT</p>	Specifies the number of elements to be calculated.
<i>a, b</i>	<p>FORTRAN 77: COMPLEX for vcmulbyconj, vmcmulbyconj</p> <p>DOUBLE COMPLEX for vzmulbyconj, vmzmulbyconj</p> <p>Fortran 90: COMPLEX, INTENT(IN) for vcmulbyconj, vmcmulbyconj</p> <p>DOUBLE COMPLEX, INTENT(IN) for vzmulbyconj, vmzmulbyconj</p> <p>C: const MKL_Complex8* for vcMulByConj, vmcMulByConj</p> <p>const MKL_Complex16* for vzMulByConj, vmzMulByConj</p>	<p>Fortran: Arrays that specify the input vectors <i>a</i> and <i>b</i>.</p> <p>C: Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i>.</p>
<i>mode</i>	<p>FORTRAN 77: INTEGER*8</p> <p>Fortran 90: INTEGER(KIND=8), INTENT(IN)</p> <p>C: const MKL_INT64</p>	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<p>FORTRAN 77: COMPLEX for vcmulbyconj, vmcmulbyconj</p> <p>DOUBLE COMPLEX for vzmulbyconj, vmzmulbyconj</p> <p>Fortran 90: COMPLEX, INTENT(OUT) for vcmulbyconj, vmcmulbyconj</p> <p>DOUBLE COMPLEX, INTENT(OUT) for vzmulbyconj, vmzmulbyconj</p> <p>C: MKL_Complex8* for vcMulByConj, vmcMulByConj</p> <p>MKL_Complex16* for vzMulByConj, vmzMulByConj</p>	<p>Fortran: Array that specifies the output vector <i>y</i>.</p> <p>C: Pointer to an array that contains the output vector <i>y</i>.</p>

Description

The `v?MulByConj` function performs element by element multiplication of vector `a` element and conjugated vector `b` element.

Specifications for special values of the functions are found according to the formula

$$\text{MulByConj}(x_1+i*y_1, x_2+i*y_2) = \text{Mul}(x_1+i*y_1, x_2-i*y_2).$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when `x1`, `x2`, `y1`, `y2` are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VML Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

`v?Conj`

Performs element by element conjugation of the vector.

Syntax

Fortran:

```
call vcconj( n, a, y )
call vmcconj( n, a, y, mode )
call vzconj( n, a, y )
call vmzconj( n, a, y, mode )
```

C:

```
vcConj( n, a, y );
vmcConj( n, a, y, mode );
vzConj( n, a, y );
vmzConj( n, a, y, mode );
```

Include Files

- Fortran: `mkl_vml.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>n</code>	<code>FORTRAN 77: INTEGER</code> <code>Fortran 90: INTEGER, INTENT(IN)</code> <code>C: const MKL_INT</code>	Specifies the number of elements to be calculated.
<code>a</code>	<code>FORTRAN 77: COMPLEX, INTENT(IN)</code> <code>for vcconj, vmcconj</code> <code>DOUBLE COMPLEX, INTENT(IN) for vzconj, vmzconj</code> <code>Fortran 90: COMPLEX, INTENT(IN)</code> <code>for vcconj, vmcconj</code>	<code>Fortran:</code> Array that specifies the input vector <code>a</code> . <code>C:</code> Pointer to an array that contains the input vector <code>a</code> .

Name	Type	Description
	DOUBLE COMPLEX, INTENT(IN) for vzconj, vmzconj C: const MKL_Complex8* for vcConj, vmcConj const MKL_Complex16* for vzConj, vmzConj	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: COMPLEX, for vcconj, vmcconj DOUBLE COMPLEX for vzconj, vmzconj Fortran 90: COMPLEX, INTENT(OUT) for vcconj, vmcconj DOUBLE COMPLEX, INTENT(OUT) for vzconj, vmzconj C: MKL_Complex8* for vcConj, vmcConj MKL_Complex16* for vzConj, vmzConj	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The v?Conj function performs element by element conjugation of the vector.

No special values are specified. The function does not raise floating-point exceptions.

v?Abs

Computes absolute value of vector elements.

Syntax

Fortran:

```
call vsabs( n, a, y )
call vmsabs( n, a, y, mode )
call vdabs( n, a, y )
call vmdabs( n, a, y, mode )
call vcabs( n, a, y )
call vmcabs( n, a, y, mode )
```

```
call vzabs( n, a, y )
call vmzabs( n, a, y, mode )
```

C:

```
vsAbs( n, a, y );
vmsAbs( n, a, y, mode );
vdAbs( n, a, y );
vmdAbs( n, a, y, mode );
vcAbs( n, a, y );
vmcAbs( n, a, y, mode );
vzAbs( n, a, y );
vmzAbs( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
a	FORTRAN 77: REAL for vsabs, vmsabs DOUBLE PRECISION for vdabs, vmdabs COMPLEX for vcabs, vmcabs DOUBLE COMPLEX for vzabs, vmzabs Fortran 90: REAL, INTENT(IN) for vsabs, vmsabs DOUBLE PRECISION, INTENT(IN) for vdabs, vmdabs COMPLEX, INTENT(IN) for vcabs, vmcabs DOUBLE COMPLEX, INTENT(IN) for vzabs, vmzabs C: const float* for vsAbs, vmsAbs const double* for vdAbs, vmdAbs const MKL_Complex8* for vcAbs, vmcAbs	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	const MKL_Complex16* for vzAbs, vmzAbs	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vsabs, vmsabs, vcabs, vmcabs DOUBLE PRECISION for vdabs, vmdabs, vzabs, vmzabs Fortran 90: REAL, INTENT(OUT) for vsabs, vmsabs, vcabs, vmcabs DOUBLE PRECISION, INTENT(OUT) for vdabs, vmdabs, vzabs, vmzabs C: float* for vsAbs, vmsAbs, vcAbs, vmcAbs double* for vdAbs, vmdAbs, vzAbs, vmzAbs	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Abs` function computes an absolute value of vector elements.

Special Values for Real Function `v?Abs(x)`

Argument	Result	Exception
+0	+0	
-0	+0	
$+\infty$	$+\infty$	
$-\infty$	$+\infty$	
QNAN	QNAN	
SNAN	QNAN	INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Abs}(z) = \text{Hypot}(\text{RE}(z), \text{IM}(z)).$$

v?Arg

Computes argument of vector elements.

Syntax

Fortran:

```
call vcarg( n, a, y )
call vmcarg( n, a, y, mode )
```

```
call vzarg( n, a, y )
call vmzarg( n, a, y, mode )
```

C:

```
vcArg( n, a, y );
vmcArg( n, a, y, mode );
vzArg( n, a, y );
vmzArg( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN77: COMPLEX for vcarg, vmcarg DOUBLE COMPLEX for vzarg, vmzarg Fortran 90: COMPLEX, INTENT(IN) for vcarg, vmcarg DOUBLE COMPLEX , INTENT(IN) for vzarg, vmzarg C: const MKL_Complex8* for vcArg, vmcArg const MKL_Complex16* for vzArg, vmcArg	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vcarg, vmcarg DOUBLE PRECISION for vzarg, vmzarg Fortran 90: REAL, INTENT(OUT) for vcarg, vmcarg	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
		DOUBLE PRECISION, INTENT(OUT) for vzarg, vmzarg C: float* for vcArg, vmcArg double* for vzArg, vmcArg

Description

The v?Arg function computes argument of vector elements.

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function v?Arg(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+3·π/4	+π/2	+π/2	+π/2	+π/2	+π/4	NAN
+i·Y	+π		+π/2	+π/2		+0	NAN
+i·0	+π	+π	+π	+0	+0	+0	NAN
-i·0	-π	-π	-π	-0	-0	-0	NAN
-i·Y	-π		-π/2	-π/2		-0	NAN
-i·∞	-3·π/4	-π/2	-π/2	-π/2	-π/2	-π/4	NAN
+i·NAN	NAN						

Notes:

- raises INVALID exception when real or imaginary part of the argument is SNAN
- $\text{Arg}(z) = \text{Atan2}(\text{IM}(z), \text{RE}(z))$.

v?LinearFrac

Performs linear fraction transformation of vectors *a* and *b* with scalar parameters.

Syntax

Fortran:

```
call vslinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y )
call vmslinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y, mode )
call vdlinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y )
call vmdlinearfrac( n, a, b, scalea, shifta, scaleb, shiftb, y, mode )
```

C:

```
vsLinearFrac( n, a, b, scalea, shifta, scaleb, shiftb, y );
vmsLinearFrac( n, a, b, scalea, shifta, scaleb, shiftb, y, mode );
vdLinearFrac( n, a, b, scalea, shifta, scaleb, shiftb, y )
vmdLinearFrac( n, a, b, scalea, shifta, scaleb, shiftb, y, mode );
```

Include Files

- Fortran: mkl_vml.f90

- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	FORTRAN 77: REAL for vslinearfrac, vmslinearfrac DOUBLE PRECISION for vdlinearfrac, vmdlinearfrac Fortran 90: REAL, INTENT(IN) for vslinearfrac, vmslinearfrac DOUBLE PRECISION, INTENT(IN) for vdlinearfrac, vmdlinearfrac C: const float* for vsLinearFrac, vmsLinearFrac const double* for vdLinearFrac, vmdLinearFrac	Fortran: Arrays that specify the input vectors <i>a</i> and <i>b</i> . C: Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>scalea, scaleb</i>	FORTRAN 77: REAL for vslinearfrac, vmslinearfrac DOUBLE PRECISION for vdlinearfrac, vmdlinearfrac Fortran 90: REAL, INTENT(IN) for vslinearfrac, vmslinearfrac DOUBLE PRECISION, INTENT(IN) for vdlinearfrac, vmdlinearfrac C: const float* for vsLinearFrac, vmsLinearFrac const double* for vdLinearFrac, vmdLinearFrac	Constant values for shifting addends of vectors <i>a</i> and <i>b</i> .
<i>shifta, shiftb</i>	FORTRAN 77: REAL for vslinearfrac, vmslinearfrac DOUBLE PRECISION for vdlinearfrac, vmdlinearfrac Fortran 90: REAL, INTENT(IN) for vslinearfrac, vmslinearfrac DOUBLE PRECISION, INTENT(IN) for vdlinearfrac, vmdlinearfrac C: const float* for vsLinearFrac, vmsLinearFrac	Constant values for scaling multipliers of vectors <i>a</i> and <i>b</i> .

Name	Type	Description
	const double* for vdLinearFrac, vmdLinearFrac	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vslinearfrac, vmslinearfrac DOUBLE PRECISION for vdlinearfrac, vmdlinearfrac Fortran 90: REAL, INTENT(OUT) for vslinearfrac, vmslinearfrac DOUBLE PRECISION, INTENT(OUT) for vdlinearfrac, vmdlinearfrac C: float* for vsLinearFrac, vmsLinearFrac double* for vdLinearFrac, vmdLinearFrac	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The `v?LinearFrac` function performs linear fraction transformation of vectors a by vector b with scalar parameters: scaling multipliers $scalea$, $scaleb$ and shifting addends $shifta$, $shiftb$:

$$y[i] = (scalea \cdot a[i] + shifta) / (scaleb \cdot b[i] + shiftb), i=1,2 \dots n$$

The `v?LinearFrac` function is implemented in the EP accuracy mode only, therefore no special values are defined for this function. If used in HA or LA mode, `v?LinearFrac` sets the VML Error Status to `VML_STATUS_ACCURACYWARNING` (see the [Values of the VML Status](#) table). Correctness is guaranteed within the threshold limitations defined for each input parameter (see the table below); otherwise, the behavior is unspecified.

Threshold Limitations on Input Parameters

$2^{E_{\text{MIN}}/2} \leq scalea \leq 2^{(E_{\text{MAX}}-2)/2}$
$2^{E_{\text{MIN}}/2} \leq scaleb \leq 2^{(E_{\text{MAX}}-2)/2}$
$ shifta \leq 2^{E_{\text{MAX}}-2}$
$ shiftb \leq 2^{E_{\text{MAX}}-2}$
$2^{E_{\text{MIN}}/2} \leq a[i] \leq 2^{(E_{\text{MAX}}-2)/2}$
$2^{E_{\text{MIN}}/2} \leq b[i] \leq 2^{(E_{\text{MAX}}-2)/2}$
$a[i] \neq - (shifta/scalea) * (1-\delta_1), \delta_1 \leq 2^{1-(p-1)/2}$
$b[i] \neq - (shiftb/scaleb) * (1-\delta_2), \delta_2 \leq 2^{1-(p-1)/2}$

E_{MIN} and E_{MAX} are the maximum and minimum exponents and p is the number of significant bits (precision) for corresponding data type according to the ANSI/IEEE Std 754-2008 standard ([[IEEE754](#)]):

- for single precision $E_{\text{MIN}} = -126$, $E_{\text{MAX}} = 127$, $p = 24$
- for double precision $E_{\text{MIN}} = -1022$, $E_{\text{MAX}} = 1023$, $p = 53$

The thresholds become less strict for common cases with $\text{scalea}=0$ and/or $\text{scaleb}=0$:

- if $\text{scalea}=0$, there are no limitations for the values of $a[i]$ and shifta
- if $\text{scaleb}=0$, there are no limitations for the values of $b[i]$ and shiftb

Power and Root Functions

v?Inv

Performs element by element inversion of the vector.

Syntax

Fortran:

```
call vsinv( n, a, y )
call vmsinv( n, a, y, mode )
call vdinv( n, a, y )
call vmdinv( n, a, y, mode )
```

C:

```
vsInv( n, a, y );
vmsInv( n, a, y, mode );
vdInv( n, a, y );
vmdInv( n, a, y, mode );
```

Include Files

- Fortran: `mkl_vml.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77 : INTEGER Fortran 90 : INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77 : REAL for <code>vsinv</code> , <code>vmsinv</code> Fortran 90 : REAL, INTENT(IN) for <code>vsinv</code> , <code>vmsinv</code>	Fortran : Array that specifies the input vector <i>a</i> . C : Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdinv, vmdinv C: const float* for vsInv, vmsInv const double* for vdInv, vmdInv	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vsinv, vmsinv DOUBLE PRECISION for vdinv, vmdinv Fortran 90: REAL, INTENT(OUT) for vsinv, vmsinv DOUBLE PRECISION, INTENT(OUT) for vdinv, vmdinv C: float* for vsInv, vmsInv double* for vdInv, vmdInv	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The `v?Inv` function performs element by element inversion of the vector.

Special Values for Real Function `v?Inv(x)`

Argument	Result	VML Error Status	Exception
+0	+∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
+∞	+0		
-∞	-0		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Div

Performs element by element division of vector *a* by vector *b*

Syntax

Fortran:

```
call vsdiv( n, a, b, y )
```

```
call vmsdiv( n, a, b, y, mode )
call vddiv( n, a, b, y )
call vmddiv( n, a, b, y, mode )
call vcdiv( n, a, b, y )
call vmcddiv( n, a, b, y, mode )
call vzdiv( n, a, b, y )
call vmzdiv( n, a, b, y, mode )
```

C:

```
vsDiv( n, a, b, y );
vmsDiv( n, a, b, y, mode );
vdDiv( n, a, b, y );
vmdDiv( n, a, b, y, mode );
vcDiv( n, a, b, y );
vmcDiv( n, a, b, y, mode );
vzDiv( n, a, b, y );
vmzDiv( n, a, b, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	FORTRAN 77: REAL for vsdiv, vmsdiv DOUBLE PRECISION for vddiv, vmddiv COMPLEX for vcdiv, vmcddiv DOUBLE COMPLEX for vzdiv, vmzdiv Fortran 90: REAL, INTENT(IN) for vsdiv, vmsdiv DOUBLE PRECISION, INTENT(IN) for vddiv, vmddiv COMPLEX, INTENT(IN) for vcdiv, vmcddiv DOUBLE COMPLEX, INTENT(IN) for vzdiv, vmzdiv	Fortran: Arrays that specify the input vectors <i>a</i> and <i>b</i> . C: Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .

Name	Type	Description
	C: const float* for vsDiv, vmsDiv const double* for vdDiv, vmdDiv const MKL_Complex8* for vcDiv, vmcDiv const MKL_Complex16* for vzDiv, vmzDiv	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Div Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < \text{abs}(b[i]) * \text{FLT_MAX}$
double precision	$\text{abs}(a[i]) < \text{abs}(b[i]) * \text{DBL_MAX}$

Precision overflow thresholds for the complex v?Div function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vsdiv, vmsdiv DOUBLE PRECISION for vddiv, vmddiv COMPLEX for vcdiv, vmcdiv DOUBLE COMPLEX for vzdiv, vmzdiv Fortran 90: REAL, INTENT(OUT) for vsdiv, vmsdiv DOUBLE PRECISION, INTENT(OUT) for vddiv, vmddiv COMPLEX, INTENT(OUT) for vcdiv, vmcdiv DOUBLE COMPLEX, INTENT(OUT) for vzdiv, vmzdiv C: float* for vsDiv, vmsDiv double* for vdDiv, vmdDiv MKL_Complex8* for vcDiv, vmcDiv MKL_Complex16* for vzDiv, vmzDiv	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The v?Div function performs element by element division of vector *a* by vector *b*.

Special values for Real Function v?Div(x)

Argument 1	Argument 2	Result	VML Error Status	Exception
X > +0	+0	+∞	VML_STATUS_SING	ZERODIVIDE
X > +0	-0	-∞	VML_STATUS_SING	ZERODIVIDE
X < +0	+0	-∞	VML_STATUS_SING	ZERODIVIDE
X < +0	-0	+∞	VML_STATUS_SING	ZERODIVIDE
+0	+0	QNAN	VML_STATUS_SING	
-0	-0	QNAN	VML_STATUS_SING	
X > +0	+∞	+0		
X > +0	-∞	-0		
+∞	+∞	QNAN		
-∞	-∞	QNAN		
QNAN	QNAN	QNAN		
SNAN	SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Div}(x_1+i*y_1, x_2+i*y_2) = (x_1+i*y_1)*(x_2-i*y_2) / (x_2*x_2+y_2*y_2).$$

Overflow in a complex function occurs when x_2+i*y_2 is not zero, x_1, x_2, y_1, y_2 are finite numbers, but the real or imaginary part of the exact result is so large that it does not fit the target precision. In that case, the function returns ∞ in that part of the result, raises the OVERFLOW exception, and sets the VML Error Status to VML_STATUS_OVERFLOW.

v?Sqrt

Computes a square root of vector elements.

Syntax**Fortran:**

```
call vssqrt( n, a, y )
call vmssqrt( n, a, y, mode )
call vdsqrt( n, a, y )
call vmdsqrt( n, a, y, mode )
call vc_sqrt( n, a, y )
call vmc_sqrt( n, a, y, mode )
call vzsqrt( n, a, y )
call vmzsqrt( n, a, y, mode )
```

C:

```
vs_sqrt( n, a, y );
vms_sqrt( n, a, y, mode );
vd_sqrt( n, a, y );
vmd_sqrt( n, a, y, mode );
vc_sqrt( n, a, y );
vmc_sqrt( n, a, y, mode );
vz_sqrt( n, a, y );
vmz_sqrt( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER, INTENT(IN) <small>C:</small> const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	<small>FORTRAN 77:</small> REAL for vssqrt, vmssqrt <small>DOUBLE PRECISION</small> for vdsqrt, vmdsqrt <small>COMPLEX</small> for vcsqrt, vmcsqrt <small>DOUBLE COMPLEX</small> for vzsqrt, vmzsqrt <small>Fortran 90:</small> REAL, INTENT(IN) for vssqrt, vmssqrt <small>DOUBLE PRECISION, INTENT(IN)</small> for vdsqrt, vmdsqrt <small>COMPLEX, INTENT(IN)</small> for vcsqrt, vmcsqrt <small>DOUBLE COMPLEX, INTENT(IN)</small> for vzsqrt, vmzsqrt <small>C:</small> const float* for vsSqrt, vmsSqrt <small>const double*</small> for vdSqrt, vmdSqrt <small>const MKL_Complex8*</small> for vcSqrt, vmcSqrt <small>const MKL_Complex16*</small> for vzSqrt, vmzSqrt	<small>Fortran:</small> Array that specifies the input vector <i>a</i> . <small>C:</small> Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	<small>FORTRAN 77:</small> INTEGER*8 <small>Fortran 90:</small> INTEGER(KIND=8), INTENT(IN) <small>C:</small> const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<small>Fortran:</small> REAL for vssqrt, vmssqrt <small>DOUBLE PRECISION</small> for vdsqrt, vmdsqrt	<small>Fortran:</small> Array that specifies the output vector <i>y</i> .

Name	Type	Description
	COMPLEX for vcsqrt, vmcsqrt	
	DOUBLE COMPLEX for vzsqrt, vmzsqrt	C: Pointer to an array that contains the output vector y .
	Fortran 90: REAL, INTENT(OUT) for vssqrt, vmssqrt	
	DOUBLE PRECISION, INTENT(OUT) for vdsqrt, vmdsqrt	
	COMPLEX, INTENT(OUT) for vcsqrt, vmcsqrt	
	DOUBLE COMPLEX, INTENT(OUT) for vzsqrt, vmzsqrt	
	C: float* for vsqrt, vmsqrt	
	double* for vd_sqrt, vmd_sqrt	
	MKL_Complex8* for vc_sqrt, vmc_sqrt	
	MKL_Complex16* for vz_sqrt, vmz_sqrt	

Description

The $v?Sqrt$ function computes a square root of vector elements.

Special Values for Real Function $v?Sqrt(x)$

Argument	Result	VML Error Status	Exception
$X < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+0		
-0	-0		
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function $v?Sqrt(z)$

RE(z) i·IM(z)	$-\infty$	$-x$	-0	$+0$	$+x$	$+\infty$	NAN
$+i\cdot\infty$	$+\infty+i\cdot\infty$						
$+i\cdot Y$	$+0+i\cdot\infty$					$+0+i\cdot 0$	QNAN+i·QNAN
$+i\cdot 0$	$+0+i\cdot\infty$		$+0+i\cdot 0$	$+0+i\cdot 0$		$+0+i\cdot 0$	QNAN+i·QNAN
$-i\cdot 0$	$+0-i\cdot\infty$		$+0-i\cdot 0$	$+0-i\cdot 0$		$+0-i\cdot 0$	QNAN+i·QNAN
$-i\cdot Y$	$+0-i\cdot\infty$					$+0-i\cdot 0$	QNAN+i·QNAN
$-i\cdot\infty$	$+\infty-i\cdot\infty$						
$+i\cdot NAN$	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	$+0+i\cdot NAN$	QNAN+i·QNAN

Notes:

- raises INVALID exception when the real or imaginary part of the argument is SNAN
- $\text{Sqrt}(\text{CONJ}(z)) = \text{CONJ}(\text{Sqrt}(z))$.

v?InvSqrt

Computes an inverse square root of vector elements.

Syntax

Fortran:

```
call vsinvsqrt( n, a, y )
call vmsinvsqrt( n, a, y, mode )
call vdinvsqrt( n, a, y )
call vmdinvsqrt( n, a, y, mode )
```

C:

```
vsInvSqrt( n, a, y );
vmsInvSqrt( n, a, y, mode );
vdInvSqrt( n, a, y );
vmdInvSqrt( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN 77: REAL for vsinvsqrt, vmsinvsqrt DOUBLE PRECISION for vdinvsqrt, vmdinvsqrt Fortran 90: REAL, INTENT(IN) for vsinvsqrt, vmsinvsqrt DOUBLE PRECISION, INTENT(IN) for vdinvsqrt, vmdinvsqrt C: const float* for vsInvSqrt, vmsInvSqrt const double* for vdInvSqrt, vmdInvSqrt	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTTRAN 77: INTEGER*8	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Name	Type	Description
	Fortran 90: INTEGER(KIND=8), INTENT(IN)	
	C: const MKL_INT64	

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vsinvsqrt, vmsinvsqrt DOUBLE PRECISION for vdinvsqrt, vmdinvsqrt Fortran 90: REAL, INTENT(OUT) for vsinvsqrt, vmsinvsqrt DOUBLE PRECISION, INTENT(OUT) for vdinvsqrt, vmdinvsqrt C: float* for vsInvSqrt, vmsInvSqrt double* for vdInvSqrt, vmdInvSqrt	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The *v?InvSqrt* function computes an inverse square root of vector elements.

Special Values for Real Function *v?InvSqrt(x)*

Argument	Result	VML Error Status	Exception
X < +0	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Cbrt

Computes a cube root of vector elements.

Syntax

Fortran:

```
call vscbrt( n, a, y )
call vmscbrt( n, a, y, mode )
call vdcbrt( n, a, y )
call vmdcbrt( n, a, y, mode )
```

C:

```
vsCbrt( n, a, y );
vmsCbrt( n, a, y, mode );
```

```
vdCbrt( n, a, y );
vmdCbrt( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77: REAL for vscbrt, vmscbrt DOUBLE PRECISION for vdcbrt, vmdcbrt Fortran 90: REAL, INTENT(IN) for vscbrt, vmscbrt DOUBLE PRECISION, INTENT(IN) for vdcbrt, vmdcbrt C: const float* for vsCbrt, vmsCbrt const double* for vdCbrt, vmdCbrt	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vscbrt, vmscbrt DOUBLE PRECISION for vdcbrt, vmdcbrt Fortran 90: REAL, INTENT(OUT) for vscbrt, vmscbrt DOUBLE PRECISION, INTENT(OUT) for vdcbrt, vmdcbrt C: float* for vsCbrt, vmsCbrt double* for vdCbrt, vmdCbrt	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Cbrt` function computes a cube root of vector elements.

Special Values for Real Function `v?Cbrt(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	INVALID

`v?InvCbrt`

Computes an inverse cube root of vector elements.

Syntax

Fortran:

```
call vsinvcbrt( n, a, y )
call vmsinvcbrt( n, a, y, mode )
call vdinvcbcrt( n, a, y )
call vmdinvcbcrt( n, a, y, mode )
```

C:

```
vsInvCbrt( n, a, y );
vmsInvCbrt( n, a, y, mode );
vdInvCbrt( n, a, y );
vmdInvCbrt( n, a, y, mode );
```

Include Files

- Fortran: `mkl_vml.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>n</code>	FORTTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) <code>C:</code> const MKL_INT	Specifies the number of elements to be calculated.
<code>a</code>	FORTTRAN 77: REAL for <code>vsinvcbrt</code> , <code>vmsinvcbrt</code> DOUBLE PRECISION for <code>vdinvcbcrt</code> , <code>vmdinvcbcrt</code> <code>Fortran 90: REAL, INTENT(IN) for</code> <code>vsinvcbrt, vmsinvcbrt</code>	<code>Fortran:</code> Array that specifies the input vector <code>a</code> . <code>C:</code> Pointer to an array that contains the input vector <code>a</code> .

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdinvcbrt, vmdinvcbrt C: const float* for vsInvCbrt, vmsInvCbrt const double* for vdInvCbrt, vmdInvCbrt	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vsinvcbrt, vmsinvcbrt DOUBLE PRECISION for vdinvcbrt, vmdinvcbrt Fortran 90: REAL, INTENT(OUT) for vsinvcbrt, vmsinvcbrt DOUBLE PRECISION, INTENT(OUT) for vdinvcbrt, vmdinvcbrt C: float* for vsInvCbrt, vmsInvCbrt double* for vdInvCbrt, vmdInvCbrt	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The `v?InvCbrt` function computes an inverse cube root of vector elements.

Special Values for Real Function `v?InvCbrt(x)`

Argument	Result	VML Error Status	Exception
+0	+∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
+∞	+0		
-∞	-0		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Pow2o3

Raises each element of a vector to the constant power 2/3.

Syntax

Fortran:

```
call vspow2o3( n, a, y )
```

```
call vmspow2o3( n, a, y, mode )
call vdpow2o3( n, a, y )
call vmdpow2o3( n, a, y, mode )
```

C:

```
vsPow2o3( n, a, y );
vmsPow2o3( n, a, y, mode );
vdPow2o3( n, a, y );
vmdPow2o3( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN 77: REAL for vspow2o3, vmspow2o3 DOUBLE PRECISION for vdpow2o3, vmdpow2o3 Fortran 90: REAL, INTENT(IN) for vspow2o3, vmspow2o3 DOUBLE PRECISION, INTENT(IN) for vdpow2o3, vmdpow2o3 C: const float* for vsPow2o3, vmsPow2o3 const double* for vdPow2o3, vmdPow2o3	Fortran: Arrays, specify the input vector <i>a</i> . C: Pointers to arrays that contain the input vector <i>a</i> .
<i>mode</i>	FORTTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTTRAN 77: REAL for vspow2o3, vmspow2o3 DOUBLE PRECISION for vdpow2o3, vmdpow2o3	Fortran: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
	Fortran 90: REAL, INTENT(OUT) for vspow2o3, vmspow2o3	
	DOUBLE PRECISION, INTENT(OUT) for vdpow2o3, vmdpow2o3	
	C: float* for vsPow2o3, vmsPow2o3 double* for vdPow2o3, vmdPow2o3	

Description

The $v^{2/3}$ function raises each element of a vector to the constant power 2/3.

Special Values for Real Function $v^{2/3}(x)$

Argument	Result	Exception
+0	+0	
-0	+0	
$+\infty$	$+\infty$	
$-\infty$	$+\infty$	
QNAN	QNAN	
SNAN	QNAN	INVALID

$v^{3/2}$

Raises each element of a vector to the constant power 3/2.

Syntax

Fortran:

```
call vspow3o2( n, a, y )
call vmspow3o2( n, a, y, mode )
call vdpow3o2( n, a, y )
call vmdpow3o2( n, a, y, mode )
```

C:

```
vsPow3o2( n, a, y );
vmsPow3o2( n, a, y, mode );
vdPow3o2( n, a, y );
vmdPow3o2( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77: REAL for vspow3o2, vmspow3o2 DOUBLE PRECISION for vdpow3o2, vmdpow3o2 Fortran 90: REAL, INTENT(IN) for vspow3o2, vmspow3o2 DOUBLE PRECISION, INTENT(IN) for vdpow3o2, vmdpow3o2 C: const float* for vsPow3o2, vmsPow3o2 const double* for vdPow3o2, vmdPow3o2	Fortran: Arrays, specify the input vector <i>a</i> . C: Pointers to arrays that contain the input vector <i>a</i> .
<i>mode</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Pow3o2 Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT_MAX})^{2/3}$
double precision	$\text{abs}(a[i]) < (\text{DBL_MAX})^{2/3}$

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vspow3o2, vmspow3o2 DOUBLE PRECISION for vdpow3o2, vmdpow3o2 Fortran 90: REAL, INTENT(OUT) for vspow3o2, vmspow3o2 DOUBLE PRECISION, INTENT(OUT) for vdpow3o2, vmdpow3o2 C: float* for vsPow3o2, vmsPow3o2 double* for vdPow3o2, vmdPow3o2	Fortran: Array, specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Pow3o2` function raises each element of a vector to the constant power 3/2.

Special Values for Real Function v?Pow3o2(x)

Argument	Result	VML Error Status	Exception
X < +0	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+0		
-0	-0		
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Pow

Computes a to the power b for elements of two vectors.

Syntax**Fortran:**

```
call vspow( n, a, b, y )
call vmspown( n, a, b, y, mode )
call vdpow( n, a, b, y )
call vmdpow( n, a, b, y, mode )
call vcpow( n, a, b, y )
call vmcpow( n, a, b, y, mode )
call vzpow( n, a, b, y )
call vmzpow( n, a, b, y, mode )
```

C:

```
vsPow( n, a, b, y );
vmsPow( n, a, b, y, mode );
vdPow( n, a, b, y );
vmdPow( n, a, b, y, mode );
vcPow( n, a, b, y );
vmcpow( n, a, b, y, mode );
vzPow( n, a, b, y );
vmzPow( n, a, b, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.

Name	Type	Description
	C: const MKL_INT	
<i>a, b</i>	FORTRAN 77: REAL for vspow, vmspow DOUBLE PRECISION for vdpow, vmdpow COMPLEX for vcpow, vmcpow DOUBLE COMPLEX for vzpow, vmzpow Fortran 90: REAL, INTENT(IN) for vspow, vmspow DOUBLE PRECISION, INTENT(IN) for vdpow, vmdpow COMPLEX, INTENT(IN) for vcPow, vmcPow DOUBLE COMPLEX, INTENT(IN) for vzPow, vmzPow C: const float* for vsPow, vmsPow const double* for vdPow, vmdPow const MKL_Complex8* for vcPow, vmcPow const MKL_Complex16* for vzPow, vmzPow	Fortran: Arrays that specify the input vectors <i>a</i> and <i>b</i> . C: Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>mode</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Pow Function

Data Type	Threshold Limitations on Input Parameters
single precision	$\text{abs}(a[i]) < (\text{FLT_MAX})^{1/b[i]}$
double precision	$\text{abs}(a[i]) < (\text{DBL_MAX})^{1/b[i]}$

Precision overflow thresholds for the complex v?Pow function are beyond the scope of this document.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vspow, vmspow DOUBLE PRECISION for vdpow, vmdpow COMPLEX for vcPow, vmcPow DOUBLE COMPLEX for vzPow, vmzPow	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
		<pre> Fortran 90: REAL, INTENT(OUT) for vspow, vmspow DOUBLE PRECISION, INTENT(OUT) for vdpow, vmdpow COMPLEX, INTENT(OUT) for vcpow, vmcpow DOUBLE COMPLEX, INTENT(OUT) for vzpow, vmzpow C: float* for vsPow, vmsPow double* for vdPow, vmdPow MKL_Complex8* for vcPow, vmcPow MKL_Complex16* for vzPow, vmzPow </pre>

Description

The `v?Pow` function computes a to the power b for elements of two vectors.

The real function `v(s/d) Pow` has certain limitations on the input range of a and b parameters. Specifically, if $a[i]$ is positive, then $b[i]$ may be arbitrary. For negative $a[i]$, the value of $b[i]$ must be an integer (either positive or negative).

The complex function `v(c/z) Pow` has no input range limitations.

Special values for Real Function `v?Pow(x)`

Argument 1	Argument 2	Result	VML Error Status	Exception
+0	neg. odd integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. odd integer	$-\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
+0	neg. even integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. even integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
+0	neg. non-integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	neg. non-integer	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	pos. odd integer	$+0$		
-0	pos. odd integer	-0		
+0	pos. even integer	$+0$		
-0	pos. even integer	$+0$		
+0	pos. non-integer	$+0$		
-0	pos. non-integer	$+0$		
-1	$+\infty$	$+1$		
-1	$-\infty$	$+1$		
+1	any value	$+1$		
+1	$+0$	$+1$		
+1	-0	$+1$		
+1	$+\infty$	$+1$		
+1	$-\infty$	$+1$		
+1	QNAN	$+1$		
any value	$+0$	$+1$		
+0	$+0$	$+1$		

Argument 1	Argument 2	Result	VML Error Status	Exception
-0	+0	+1		
$+\infty$	+0	+1		
$-\infty$	+0	+1		
QNAN	+0	+1		
any value	-0	+1		
+0	-0	+1		
-0	-0	+1		
$+\infty$	-0	+1		
$-\infty$	-0	+1		
QNAN	-0	+1		
$X < +0$	non-integer	QNAN	VML_STATUS_ERRDOM	INVALID
$ X < 1$	$-\infty$	$+\infty$		
+0	$-\infty$	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
-0	$-\infty$	$+\infty$	VML_STATUS_ERRDOM	ZERODIVIDE
$ X > 1$	$-\infty$	+0		
$+\infty$	$-\infty$	+0		
$-\infty$	$-\infty$	+0		
$ X < 1$	$+\infty$	+0		
+0	$+\infty$	+0		
-0	$+\infty$	+0		
$ X > 1$	$+\infty$	$+\infty$		
$+\infty$	$+\infty$	$+\infty$		
$-\infty$	$+\infty$	$+\infty$		
$-\infty$	neg. odd integer	-0		
$-\infty$	neg. even integer	+0		
$-\infty$	neg. non-integer	+0		
$-\infty$	pos. odd integer	$-\infty$		
$-\infty$	pos. even integer	$+\infty$		
$-\infty$	pos. non-integer	$+\infty$		
$+\infty$	$X < +0$	+0		
$+\infty$	$X > +0$	$+\infty$		
QNAN	QNAN	QNAN		
QNAN	SNAN	QNAN		INVALID
SNAN	QNAN	QNAN		INVALID
SNAN	SNAN	QNAN		INVALID

The complex double precision versions of this function, `vzPow` and `vmzPow`, are implemented in the EP accuracy mode only. If used in HA or LA mode, `vzPow` and `vmzPow` set the VML Error Status to `VML_STATUS_ACCURACYWARNING` (see the [Values of the VML Status](#) table).

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when x_1, x_2, y_1, y_2 are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns ∞ in that part of the result, raises the `OVERFLOW` exception, and sets the VML Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

v?Powx

Raises each element of a vector to the constant power.

Syntax

Fortran:

```
call vspowx( n, a, b, y )
call vmspowl( n, a, b, y, mode )
call vdpoew( n, a, b, y )
call vmdpoew( n, a, b, y, mode )
call vcpoew( n, a, b, y )
call vmcpoew( n, a, b, y, mode )
call vzpoew( n, a, b, y )
call vmzpoew( n, a, b, y, mode )
```

C:

```
vsPowx( n, a, b, y );
vmsPowx( n, a, b, y, mode );
vdPowx( n, a, b, y );
vmdPowx( n, a, b, y, mode );
vcPowx( n, a, b, y );
vmcPowx( n, a, b, y, mode );
vzPowx( n, a, b, y );
vmzPowx( n, a, b, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Number of elements to be calculated.
a	FORTAN 77: REAL for vspowx, vmspowl DOUBLE PRECISION for vdpoew, vmdpoew COMPLEX for vcpoew, vmcpoew DOUBLE COMPLEX for vzpoew, vmzpoew	Fortran: Array a that specifies the input vector C: Pointer to an array that contains the input vector a.

Name	Type	Description
	<pre> Fortran 90: REAL, INTENT(IN) for vspowx, vmspox DOUBLE PRECISION, INTENT(IN) for vdpowx, vmdpowx COMPLEX, INTENT(IN) for vcpowx, vmcpowx DOUBLE COMPLEX, INTENT(IN) for vzpowx, vmzpowx C: const float* for vsPowx, vmsPowx const double* for vdPowx, vmdPowx const MKL_Complex8* for vcPowx, vmcPowx const MKL_Complex16* for vzPowx, vmzPowx </pre>	
<i>b</i>	<pre> FORTRAN 77: REAL for vspowx, vmspox DOUBLE PRECISION for vdpowx, vmdpowx COMPLEX for vcpowx, vmcpowx DOUBLE COMPLEX for vzpowx, vmzpowx Fortran 90: REAL, INTENT(IN) for vspowx, vmspox DOUBLE PRECISION, INTENT(IN) for vdpowx, vmdpowx COMPLEX, INTENT(IN) for vcpowx, vmcpowx DOUBLE COMPLEX, INTENT(IN) for vzpowx, vmzpowx C: const float for vsPowx, vmsPowx const double for vdPowx, vmdPowx const MKL_Complex8 for vcPowx, vmcPowx const MKL_Complex16 for vzPowx, vmzPowx </pre>	<p>Fortran: Scalar value <i>b</i> that is the constant power.</p> <p>C: Constant value for power <i>b</i>.</p>
<i>mode</i>	<pre> FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) </pre>	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Name	Type	Description
	C: const MKL_INT64	

Precision Overflow Thresholds for Real v?Powx Function

Data Type	Threshold Limitations on Input Parameters
single precision	abs(a[i]) < (FLT_MAX) ^{1/b}
double precision	abs(a[i]) < (DBL_MAX) ^{1/b}

Precision overflow thresholds for the complex v?Powx function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vspowx, vmspowx DOUBLE PRECISION for vdpowx, vmdpowx COMPLEX for vcpowx, vmcpowx DOUBLE COMPLEX for vzpowx, vmzpowx Fortran 90: REAL, INTENT(OUT) for vspowx, vmspowx DOUBLE PRECISION, INTENT(OUT) for vdpowx, vmdpowx COMPLEX, INTENT(OUT) for vcpowx, vmcpowx DOUBLE COMPLEX, INTENT(OUT) for vzpowx, vmzpowx C: float* for vsPowx, vmsPowx double* for vdPowx, vmdPowx MKL_Complex8* for vcPowx, vmcPowx MKL_Complex16* for vzPowx, vmzPowx	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The v?Powx function raises each element of a vector to the constant power.

The real function v(s/d) Powx has certain limitations on the input range of a and b parameters. Specifically, if $a[i]$ is positive, then b may be arbitrary. For negative $a[i]$, the value of b must be an integer (either positive or negative).

The complex function v(c/z) Powx has no input range limitations.

Special values and VML Error Status treatment are the same as for the v?Pow function.

v?Hypot

Computes a square root of sum of two squared elements.

Syntax

Fortran:

```
call vshypot( n, a, b, y )
call vmshypot( n, a, b, y, mode )
call vdhypot( n, a, b, y )
call vmdhypot( n, a, b, y, mode )
```

C:

```
vsHypot( n, a, b, y );
vmsHypot( n, a, b, y, mode );
vdHypot( n, a, b, y );
vmdHypot( n, a, b, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER, INTENT(IN) <small>C:</small> const MKL_INT	Number of elements to be calculated.
<i>a, b</i>	<small>FORTRAN 77:</small> REAL for vshypot, vmshypot <small>DOUBLE PRECISION for vdhypot, vmdhypot</small> <small>Fortran 90:</small> REAL, INTENT(IN) for vshypot, vmshypot <small>DOUBLE PRECISION, INTENT(IN) for vdhypot, vmdhypot</small> <small>C:</small> const float* for vsHypot, vmsHypot const double* for vdHypot, vmdHypot	<small>Fortran:</small> Arrays that specify the input vectors <i>a</i> and <i>b</i> <small>C:</small> Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>mode</i>	<small>FORTRAN 77:</small> INTEGER*8 <small>Fortran 90:</small> INTEGER(KIND=8), INTENT(IN) <small>C:</small> const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Hypot Function

Data Type	Threshold Limitations on Input Parameters
single precision	<code>abs(a[i]) < sqrt(FLT_MAX)</code> <code>abs(b[i]) < sqrt(FLT_MAX)</code>
double precision	<code>abs(a[i]) < sqrt(DBL_MAX)</code> <code>abs(b[i]) < sqrt(DBL_MAX)</code>

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vshypot, vmshypot DOUBLE PRECISION for vdhypot, vmdhypot Fortran 90: REAL, INTENT(OUT) for vshypot, vmshypot DOUBLE PRECISION, INTENT(OUT) for vdhypot, vmdhypot C: float* for vsHypot, vmsHypot double* for vdHypot, vmdHypot	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The function v?Hypot computes a square root of sum of two squared elements.

Special values for Real Function v?Hypot(x)

Argument 1	Argument 2	Result	Exception
+0	+0	+0	
-0	-0	+0	
$+\infty$	any value	$+\infty$	
any value	$+\infty$	$+\infty$	
SNAN	any value	QNAN	INVALID
any value	SNAN	QNAN	INVALID
QNAN	any value	QNAN	
any value	QNAN	QNAN	

Exponential and Logarithmic Functions

v?Exp

Computes an exponential of vector elements.

Syntax

Fortran:

```
call vsexp( n, a, y )
call vmsexp( n, a, y, mode )
```

```
call vdexp( n, a, y )
call vmdexp( n, a, y, mode )
call vcexp( n, a, y )
call vmcexp( n, a, y, mode )
call vzexp( n, a, y )
call vmzexp( n, a, y, mode )
```

C:

```
vsExp( n, a, y );
vmsExp( n, a, y, mode );
vdExp( n, a, y );
vmdExp( n, a, y, mode );
vcExp( n, a, y );
vmcExp( n, a, y, mode );
vzExp( n, a, y );
vmzExp( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER, INTENT(IN) <small>C:</small> const MKL_INT	Specifies the number of elements to be calculated.
a	<small>FORTRAN 77:</small> REAL for vsexp, <small>vmsexp</small> <small>DOUBLE PRECISION for vdexp,</small> <small>vmdexp</small> <small>COMPLEX for vcexp, vmcexp</small> <small>DOUBLE COMPLEX for vzexp, vmzexp</small> <small>Fortran 90:</small> REAL, INTENT(IN) for <small>vsexp, vmsexp</small> <small>DOUBLE PRECISION, INTENT(IN) for</small> <small>vdexp, vmdexp</small> <small>COMPLEX, INTENT(IN) for vcexp,</small> <small>vmcexp</small> <small>DOUBLE COMPLEX, INTENT(IN) for</small> <small>vzexp, vmzexp</small> <small>C: const float* for vsExp, vmsExp</small>	<small>Fortran:</small> Array, specifies the input vector <i>a</i> . <small>C:</small> Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	const double* for vdExp, vmdExp const MKL_Complex8* for vcExp, vmcExp const MKL_Complex16* for vzExp, vmzExp	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Exp Function

Data Type	Threshold Limitations on Input Parameters
single precision	$a[i] < \ln(\text{FLT_MAX})$
double precision	$a[i] < \ln(\text{DBL_MAX})$

Precision overflow thresholds for the complex v?Exp function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vsexp, vmsexp DOUBLE PRECISION for vdexp, vmdexp COMPLEX for vcexp, vmcexp DOUBLE COMPLEX for vzexp, vmzexp Fortran 90: REAL, INTENT(OUT) for vsexp, vmsexp DOUBLE PRECISION, INTENT(OUT) for vdexp, vmdexp COMPLEX, INTENT(OUT) for vcexp, vmcexp DOUBLE COMPLEX, INTENT(OUT) for vzexp, vmzexp C: float* for vsExp, vmsExp double* for vdExp, vmdExp MKL_Complex8* for vcExp, vmcExp MKL_Complex16* for vzExp, vmzExp	Fortran: Array, specifies the output vector y . C: Pointer to an array that contains the output vector y .

Description

The v?Exp function computes an exponential of vector elements.

Special Values for Real Function v?Exp(x)

Argument	Result	VML Error Status	Exception
+0	+1		
-0	+1		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
X < underflow	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
+∞	+∞		
-∞	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function v?Exp(z)

RE(z)	-∞	-X	-0	+0	+X	+∞	NAN
i·IM(z)							
+i·∞	+0+i·0	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	+∞+i·QNAN INVALID	QNAN+i·QNAN INVALID
+i·Y	+0·CIS(Y)					+∞·CIS(Y)	QNAN+i·QNAN
+i·0	+0·CIS(0)		+1+i·0	+1+i·0		+∞+i·0	QNAN+i·0
-i·0	+0·CIS(0)		+1-i·0	+1-i·0		+∞-i·0	QNAN-i·0
-i·Y	+0·CIS(Y)					+∞·CIS(Y)	QNAN+i·QNAN
-i·∞	+0-i·0	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	+∞+i·QNAN INVALID	QNAN+i·QNAN
+i·NAN	+0+i·0	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	+∞+i·QNAN	QNAN+i·QNAN

Notes:

- raises the `INVALID` exception when real or imaginary part of the argument is SNAN
- raises the `INVALID` exception on argument $z = -\infty + i \cdot \text{QNAN}$
- raises the `OVERFLOW` exception and sets the VML Error Status to `VML_STATUS_OVERFLOW` in the case of overflow, that is, when $\text{RE}(z)$, $\text{IM}(z)$ are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.

v?Expm1

Computes an exponential of vector elements decreased by 1.

Syntax**Fortran:**

```
call vsexpml( n, a, y )
call vmsexpml( n, a, y, mode )
call vdexpml( n, a, y )
call vdexpml( n, a, y, mode )
```

C:

```
vsExpml( n, a, y );
```

```
vmsExpm1( n, a, y, mode );
vdExpm1( n, a, y );
vmdExpm1( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77: REAL for vsexpm1, vmsexpm1 DOUBLE PRECISION for vdexpm1, vmdexpm1 Fortran 90: REAL, INTENT(IN) for vsexpm1, vmsexpm1 DOUBLE PRECISION, INTENT(IN) for vdexpm1, vmdexpm1 C: const float* for vsExpm1, vmsExpm1 const double* for vdExpm1, vmdExpm1	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Expm1 Function

Data Type	Threshold Limitations on Input Parameters
single precision	$a[i] < \ln(\text{FLT_MAX})$
double precision	$a[i] < \ln(\text{DBL_MAX})$

Output Parameters

Name	Type	Description
<i>y</i>	FORTAN 77: REAL for vsexpm1, vmsexpm1 DOUBLE PRECISION for vdexpm1, vmdexpm1	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
		Fortran 90: REAL, INTENT(OUT) for vsexpm1, vmsexpm1
		DOUBLE PRECISION, INTENT(OUT) for vdexpm1, vmdexpm1
		C: float* for vsExpm1, vmsExpm1 double* for vdExpm1, vmdExpm1

Description

The $v\text{?Expm1}$ function computes an exponential of vector elements decreased by 1.

Special Values for Real Function $v\text{?Expm1}(x)$

Argument	Result	VML Error Status	Exception
+0	+0		
-0	+0		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
+∞	+∞		
-∞	-1		
QNAN	QNAN		
SNAN	QNAN		INVALID

v?Ln

Computes natural logarithm of vector elements.

Syntax

Fortran:

```
call vsln( n, a, y )
call vmsln( n, a, y, mode )
call vdln( n, a, y )
call vmdln( n, a, y, mode )
call vcln( n, a, y )
call vmcln( n, a, y, mode )
call vzln( n, a, y )
call vmzln( n, a, y, mode )
```

C:

```
vsLn( n, a, y );
vmsLn( n, a, y, mode );
vdLn( n, a, y );
vmdLn( n, a, y, mode );
vcLn( n, a, y );
vmcln( n, a, y, mode );
vzLn( n, a, y );
```

```
vmzLn( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER, INTENT(IN)</p> <p>C: const MKL_INT</p>	Specifies the number of elements to be calculated.
<i>a</i>	<p>FORTRAN 77: REAL for vsln, vmsln</p> <p>DOUBLE PRECISION for vdln, vmdln</p> <p>COMPLEX for vcln, vmcln</p> <p>DOUBLE COMPLEX for vzln, vmzln</p> <p>Fortran 90: REAL, INTENT(IN) for vsln, vmsln</p> <p>DOUBLE PRECISION, INTENT(IN) for vdln, vmdln</p> <p>COMPLEX, INTENT(IN) for vcln, vmcln</p> <p>DOUBLE COMPLEX, INTENT(IN) for vzln, vmzln</p> <p>C: const float* for vsln, vmsln</p> <p>const double* for vdln, vmdln</p> <p>const MKL_Complex8* for vcln, vmcln</p> <p>const MKL_Complex16* for vzln, vmzln</p>	<p>Fortran: Array that specifies the input vector <i>a</i>.</p> <p>C: Pointer to an array that contains the input vector <i>a</i>.</p>
<i>mode</i>	<p>FORTRAN 77: INTEGER*8</p> <p>Fortran 90: INTEGER(KIND=8), INTENT(IN)</p> <p>C: const MKL_INT64</p>	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<p>FORTRAN 77: REAL for vsln, vmsln</p> <p>DOUBLE PRECISION for vdln, vmdln</p> <p>COMPLEX for vcln, vmcln</p> <p>DOUBLE COMPLEX for vzln, vmzln</p>	<p>Fortran: Array that specifies the output vector <i>y</i>.</p> <p>C: Pointer to an array that contains the output vector <i>y</i>.</p>

Name	Type	Description
	Fortran 90: REAL, INTENT(OUT) for vsln, vmsln	
	DOUBLE PRECISION, INTENT(OUT) for vdln, vmdln	
	COMPLEX, INTENT(OUT) for vcLn, vmcLn	
	DOUBLE COMPLEX, INTENT(OUT) for vzLn, vmzLn	
	C: float* for vsLn, vmsLn double* for vdLn, vmdLn	
	MKL_Complex8* for vcLn, vmcLn	
	MKL_Complex16* for vzLn, vmzLn	

Description

The $v?Ln$ function computes natural logarithm of vector elements.

Special Values for Real Function $v?Ln(x)$

Argument	Result	VML Error Status	Exception
+1	+0		
$X < +0$	QNAN	VML_STATUS_ERRDOM	INVALID
+0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function $v?Ln(z)$

$RE(z)$ $i \cdot IM(z)$	$-\infty$	$-x$	-0	$+0$	$+x$	$+\infty$	NAN
$+i \cdot \infty$	$+\infty + i \cdot \frac{3\pi}{4}$	$+\infty + i \cdot \pi/2$	$+\infty + i \cdot \pi/2$	$+\infty + i \cdot \pi/2$	$+\infty + i \cdot \pi/2$	$+\infty + i \cdot \pi/4$	$+\infty + i \cdot \text{QNAN}$
$+i \cdot Y$	$+\infty + i \cdot \pi$					$+\infty + i \cdot 0$	QNAN + i · QNAN INVALID
$+i \cdot 0$	$+\infty + i \cdot \pi$		$-\infty + i \cdot \pi$ ZERODIVID E	$-\infty + i \cdot 0$ ZERODIVID E		$+\infty + i \cdot 0$	QNAN + i · QNAN INVALID
$-i \cdot 0$	$+\infty - i \cdot \pi$		$-\infty - i \cdot \pi$ ZERODIVID E	$-\infty - i \cdot 0$ ZERODIVID E		$+\infty - i \cdot 0$	QNAN + i · QNAN INVALID
$-i \cdot Y$	$+\infty - i \cdot \pi$					$+\infty - i \cdot 0$	QNAN + i · QNAN INVALID

RE(z) i·IM(z)	-∞	-x	-0	+0	+x	+∞	NAN
$-i\infty$	$+\infty - i \cdot \frac{3\pi}{4}$	$+\infty - i\cdot\pi/2$	$+\infty - i\cdot\pi/2$	$+\infty - i\cdot\pi/2$	$+\infty - i\cdot\pi/2$	$+\infty - i\cdot\pi/4$	$+\infty + i\cdot\text{QNAN}$
$+i\cdot\text{NAN}$	$+\infty + i\cdot\text{QNAN}$ INVALID	$\text{QNAN} + i\cdot\text{QNAN}$ INVALID	$\text{QNAN} + i\cdot\text{QNAN}$ INVALID	$\text{QNAN} + i\cdot\text{QNAN}$ INVALID	$\text{QNAN} + i\cdot\text{QNAN}$ INVALID	$+\infty + i\cdot\text{QNAN}$ INVALID	$\text{QNAN} + i\cdot\text{QNAN}$ INVALID

Notes:

- raises INVALID exception when real or imaginary part of the argument is SNAN

v?Log10

Computes denary logarithm of vector elements.

Syntax

Fortran:

```
call vslog10( n, a, y )
call vmslog10( n, a, y, mode )
call vdlog10( n, a, y )
call vmdlog10( n, a, y, mode )
call vclog10( n, a, y )
call vmclog10( n, a, y, mode )
call vzlog10( n, a, y )
call vmzlog10( n, a, y, mode )
```

C:

```
vsLog10( n, a, y );
vmsLog10( n, a, y, mode );
vdLog10( n, a, y );
vmdLog10( n, a, y, mode );
vcLog10( n, a, y );
vmcLog10( n, a, y, mode );
vzLog10( n, a, y );
vmzLog10( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77: REAL for vslog10, vmslog10 DOUBLE PRECISION for vdlog10, vmdlog10 COMPLEX for vclog10, vmclog10 DOUBLE COMPLEX for vzlog10, vmzlog10 Fortran 90: REAL, INTENT(IN) for vslog10, vmslog10 DOUBLE PRECISION, INTENT(IN) for vdlog10, vmdlog10 COMPLEX, INTENT(IN) for vclog10, vmclog10 DOUBLE COMPLEX, INTENT(IN) for vzlog10, vmzlog10 C: const float* for vsLog10, vmsLog10 const double* for vdLog10, vmdLog10 const MKL_Complex8* for vcLog10, vmcLog10 const MKL_Complex16* for vzLog10, vmzLog10	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vslog10, vmslog10 DOUBLE PRECISION for vdlog10, vmdlog10 COMPLEX for vclog10, vmclog10	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
	DOUBLE COMPLEX for vzlog10, vmzlog10	
	Fortran 90: REAL, INTENT(OUT) for vslog10, vmslog10	
	DOUBLE PRECISION, INTENT(OUT) for vdlog10, vmdlog10	
	COMPLEX, INTENT(OUT) for vclog10, vmclog10	
	DOUBLE COMPLEX, INTENT(OUT) for vzlog10, vmzlog10	
C:	float* for vsLog10, vmsLog10	
	double* for vdLog10, vmdLog10	
	MKL_Complex8* for vcLog10, vmcLog10	
	MKL_Complex16* for vzLog10, vmzLog10	

Description

The v?Log10 function computes a denary logarithm of vector elements.

Special Values for Real Function v?Log10(x)

Argument	Result	VML Error Status	Exception
+1	+0		
X < +0	QNAN	VML_STATUS_ERRDOM	INVALID
+0	-∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function v?Log10(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\infty + i \frac{3}{4} \frac{\pi}{\ln(10)}$	$+\infty + i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty + i \frac{\pi}{4} \frac{1}{\ln(10)}$	+∞+i·QNAN INVALID			
+i·Y	$+\infty + i \frac{\pi}{\ln(10)}$					$+\infty + i \cdot 0$	QNAN+i·QNAN INVALID
+i·0	$+\infty + i \frac{\pi}{\ln(10)}$		$-\infty + i \frac{\pi}{\ln(10)}$	$-\infty + i \cdot 0$ ZERODRIVE		$+\infty + i \cdot 0$	QNAN+i·QNAN INVALID

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
-i·0	$+\infty - i \frac{\pi}{\ln(10)}$		$-\infty - i \frac{\pi}{\ln(10)}$ ZERODIVID E	$-\infty-i·0$ ZERODIVID E		$+\infty-i·0$	QNAN-i·QNAN INVALID
-i·Y	$+\infty - i \frac{\pi}{\ln(10)}$					$+\infty-i·0$	QNAN+i·QNAN INVALID
-i·∞	$+\infty + i \frac{3}{4} \frac{\pi}{\ln(10)}$	$+\infty - i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty - i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty - i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty - i \frac{\pi}{2} \frac{1}{\ln(10)}$	$+\infty - i \frac{\pi}{4} \frac{1}{\ln(10)}$	$+\infty+i·QNAN$
+i·NAN	$+\infty+i·QNAN$	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	$+\infty+i·QNAN$	QNAN+i·QNAN INVALID

Notes:

- raises INVALID exception when real or imaginary part of the argument is SNAN

v?Log1p

Computes a natural logarithm of vector elements that are increased by 1.

Syntax

Fortran:

```
call vslog1p( n, a, y )
call vmslog1p( n, a, y, mode )
call vdlog1p( n, a, y )
call vmdlog1p( n, a, y, mode )
```

C:

```
vsLog1p( n, a, y );
vmsLog1p( n, a, y, mode );
vdLog1p( n, a, y );
vmdLog1p( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.

Name	Type	Description
a	<p>FORTRAN 77: REAL for vslog1p, vmslog1p</p> <p>DOUBLE PRECISION for vdlog1p, vmdlog1p</p> <p>Fortran 90: REAL, INTENT(IN) for vslog1p, vmslog1p</p> <p>DOUBLE PRECISION, INTENT(IN) for vdlog1p, vmdlog1p</p> <p>C: const float* for vsLog1p, vmsLog1p</p> <p>const double* for vdLog1p, vmdLog1p</p>	<p>Fortran: Array that specifies the input vector a.</p> <p>C: Pointer to an array that contains the input vector a.</p>
mode	<p>FORTRAN 77: INTEGER*8</p> <p>Fortran 90: INTEGER(KIND=8), INTENT(IN)</p> <p>C: const MKL_INT64</p>	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	<p>FORTRAN 77: REAL for vslog1p, vmslog1p</p> <p>DOUBLE PRECISION for vdlog1p, vmdlog1p</p> <p>Fortran 90: REAL, INTENT(OUT) for vslog1p, vmslog1p</p> <p>DOUBLE PRECISION, INTENT(OUT) for vdlog1p, vmdlog1p</p> <p>C: float* for vsLog1p, vmsLog1p</p> <p>double* for vdLog1p, vmdLog1p</p>	<p>Fortran: Array that specifies the output vector y.</p> <p>C: Pointer to an array that contains the output vector y.</p>

Description

The `v?Log1p` function computes a natural logarithm of vector elements that are increased by 1.

Special Values for Real Function `v?Log1p(x)`

Argument	Result	VML Error Status	Exception
-1	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$X < -1$	QNAN	VML_STATUS_ERRDOM	INVALID
+0	+0		
-0	-0		
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
QNAN	QNAN		

Argument	Result	VML Error Status	Exception
SNAN	QNAN		INVALID

Trigonometric Functions

v?Cos

Computes cosine of vector elements.

Syntax

Fortran:

```
call vscos( n, a, y )
call vmscos( n, a, y, mode )
call vdcos( n, a, y )
call vmdcos( n, a, y, mode )
call vccos( n, a, y )
call vmccos( n, a, y, mode )
call vzcos( n, a, y )
call vmzcos( n, a, y, mode )
```

C:

```
vsCos( n, a, y );
vmsCos( n, a, y, mode );
vdCos( n, a, y );
vmdCos( n, a, y, mode );
vcCos( n, a, y );
vmcCos( n, a, y, mode );
vzCos( n, a, y );
vmzCos( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
a	FORTRAN 77: REAL for vscos, vmscos	Fortran: Array that specifies the input vector a.

Name	Type	Description
	DOUBLE PRECISION for vdcos, vmdcos	C: Pointer to an array that contains the input vector a .
	COMPLEX for vccos, vmccos	
	DOUBLE PRECISION for vzcos, vmzcos	
	Fortran 90: REAL, INTENT (IN) for vscos, vmscos	
	DOUBLE PRECISION, INTENT (IN) for vdcos, vmdcos	
	COMPLEX, INTENT (IN) for vccos, vmccos	
	DOUBLE PRECISION, INTENT (IN) for vzcos, vmzcos	
	C: const float* for vsCos, vmsCos const double* for vdCos, vmdCos const MKL_Complex8* for vcCos, vmcCos const MKL_Complex16* for vzCos, vmzCos	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT (IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vscos, vmscos	Fortran: Array that specifies the output vector y .
	DOUBLE PRECISION for vdcos, vmdcos	C: Pointer to an array that contains the output vector y .
	COMPLEX for vccos, vmccos	
	DOUBLE PRECISION for vzcos, vmzcos	
	Fortran 90: REAL, INTENT (OUT) for vscos, vmscos	
	DOUBLE PRECISION, INTENT (OUT) for vdcos, vmdcos	
	COMPLEX, INTENT (OUT) for vccos, vmccos	

Name	Type	Description
	DOUBLE PRECISION, INTENT(OUT) for vzcos, vmzcos C: float* for vsCos, vmsCos double* for vdCos, vmdCos MKL_Complex8* for vcCos, vmcCos MKL_Complex16* for vzCos, vmzCos	

Description

The v?Cos function computes cosine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VML provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VML High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VML Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?Cos(x)

Argument	Result	VML Error Status	Exception
+0	+1		
-0	+1		
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Cos}(z) = \text{Cosh}(i*z).$$

v?Sin

Computes sine of vector elements.

Syntax

Fortran:

```
call vssin( n, a, y )
call vmssin( n, a, y, mode )
call vdsin( n, a, y )
call vmdsin( n, a, y, mode )
call vcsin( n, a, y )
call vmcsin( n, a, y, mode )
call vzsin( n, a, y )
call vmzsin( n, a, y, mode )
```

C:

```
vsSin( n, a, y );
vmsSin( n, a, y, mode );
```

```
vdSin( n, a, y );
vmdSin( n, a, y, mode );
vcSin( n, a, y );
vmcSin( n, a, y, mode );
vzSin( n, a, y );
vmzSin( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77: REAL for vssin, vmssin DOUBLE PRECISION for vdsin, vmdsin COMPLEX for vcsin, vmcsin DOUBLE PRECISION for vzsin, vmzsin Fortran 90: REAL, INTENT(IN) for vssin, vmssin DOUBLE PRECISION, INTENT(IN) for vdsin, vmdsin COMPLEX, INTENT(IN) for vcsin, vmcsin DOUBLE PRECISION, INTENT(IN) for vzsin, vmzsin C: const float* for vsSin, vmsSin const double* for vdSin, vmdSin const MKL_Complex8* for vcSin, vmcSin const MKL_Complex16* for vzSin, vmzSin	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vssin, vmssin DOUBLE PRECISION for vdsin, vmdsin COMPLEX for vcsin, vmcsin DOUBLE PRECISION for vzsin, vmzsin Fortran 90: REAL, INTENT(OUT) for vssin, vmssin DOUBLE PRECISION, INTENT(OUT) for vdsin, vmdsin COMPLEX, INTENT(OUT) for vcsin, vmcsin DOUBLE PRECISION, INTENT(OUT) for vzsin, vmzsin C: float* for vsSin, vmsSin double* for vdSin, vmdSin MKL_Complex8* for vcSin, vmcSin MKL_Complex16* for vzSin, vmzSin	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The function computes sine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VML provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VML High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VML Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?Sin(x)

Argument	Result	VML Error Status	Exception
+0	+0		
-0	-0		
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Sin}(z) = -i * \text{Sinh}(i * z).$$

v?SinCos

Computes sine and cosine of vector elements.

Syntax

Fortran:

```
call vssincos( n, a, y, z )
call vmssincos( n, a, y, z, mode )
call vdsincos( n, a, y, z )
call vmdsincos( n, a, y, z, mode )
```

C:

```
vsSinCos( n, a, y, z );
vmsSinCos( n, a, y, z, mode );
vdSinCos( n, a, y, z );
vmdSinCos( n, a, y, z, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN 77: REAL for vssincos, vmssincos DOUBLE PRECISION for vdsincos, vmdsincos Fortran 90: REAL, INTENT(IN) for vssincos, vmssincos DOUBLE PRECISION, INTENT(IN) for vdsincos, vmdsincos C: const float* for vsSinCos, vmsSinCos const double* for vdSinCos, vmdSinCos	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y, z	<p>FORTRAN 77: REAL for vssincos, vmssincos</p> <p>DOUBLE PRECISION for vdsincos, vmdsincos</p> <p>Fortran 90: REAL, INTENT(OUT) for vssincos, vmssincos</p> <p>DOUBLE PRECISION, INTENT(OUT) for vdsincos, vmdsincos</p> <p>C: float* for vsSinCos, vmsSinCos double* for vdSinCos, vmdSinCos</p>	<p>Fortran: Arrays that specify the output vectors y (for sine values) and z (for cosine values).</p> <p>C: Pointers to arrays that contain the output vectors y (for sinevalues) and z(for cosine values).</p>

Description

The function computes sine and cosine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VML provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VML High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VML Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?SinCos(x)

Argument	Result 1	Result 2	VML Error Status	Exception
+0	+0	+1		
-0	-0	+1		
$+\infty$	QNAN	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN	QNAN		
SNAN	QNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Sin}(z) = -i * \text{Sinh}(i * z).$$

v?CIS

Computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

Syntax

Fortran:

```
call vccis( n, a, y )
call vmccis( n, a, y, mode )
call vzcis( n, a, y )
call vmzcis( n, a, y, mode )
```

C:

```
vcCIS( n, a, y );
```

```
vmcCIS( n, a, y, mode );
vzCIS( n, a, y );
vmzCIS( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77: REAL for vccis, vmccis DOUBLE PRECISION for vzcis, vmzcis Fortran 90: REAL, INTENT(IN) for vccis, vmccis DOUBLE PRECISION, INTENT(IN) for vzcis, vmzcis C: const float* for vcCIS, vmcCIS const double* for vzCIS, vmzCIS	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: COMPLEX for vccis, vmccis DOUBLE COMPLEX for vzcis, vmzcis Fortran 90: COMPLEX, INTENT(OUT) for vccis, vmccis DOUBLE COMPLEX, INTENT(OUT) for vzcis, vmzcis C: MKL_Complex8* for vcCIS, vmcCIS MKL_Complex16* for vzCIS, vmzCIS	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?CIS` function computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function `v?CIS(x)`

x	CIS(x)
+ ∞	QNAN+i·QNAN INVALID
+ 0	+1+i·0
- 0	+1-i·0
- ∞	QNAN+i·QNAN INVALID
NAN	QNAN+i·QNAN

Notes:

- raises `INVALID` exception when the argument is `SNAN`
- raises `INVALID` exception and sets the VML Error Status to `VML_STATUS_ERRDOM` for $x=+\infty, x=-\infty$

`v?Tan`

Computes tangent of vector elements.

Syntax

Fortran:

```
call vstan( n, a, y )
call vmstan( n, a, y, mode )
call vdtan( n, a, y )
call vmdtan( n, a, y, mode )
call vctan( n, a, y )
call vmctan( n, a, y, mode )
call vztan( n, a, y )
call vmztan( n, a, y, mode )
```

C:

```
vsTan( n, a, y );
vmsTan( n, a, y, mode );
vdTan( n, a, y );
vmdTan( n, a, y, mode );
vcTan( n, a, y );
vmcTan( n, a, y, mode );
vzTan( n, a, y );
```

```
vmzTan( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
a	FORTRAN 77: REAL for vstan, vmstan DOUBLE PRECISION for vdtan, vmdtan COMPLEX for vctan, vmctan DOUBLE COMPLEX for vztan, vmztan Fortran 90: REAL, INTENT(IN) for vstan, vmstan DOUBLE PRECISION, INTENT(IN) for vdtan, vmdtan COMPLEX, INTENT(IN) for vctan, vmctan DOUBLE COMPLEX, INTENT(IN) for vztan, vmztan C: const float* for vsTan, vmsTan const double* for vdTan, vmdTan const MKL_Complex8* for vcTan, vmcTan const MKL_Complex16* for vzTan, vmzTan	Fortran: Array that specifies the input vector a. C: Pointer to an array that contains the input vector a.
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vstan, vmstan	Fortran: Array that specifies the output vector y.

Name	Type	Description
	DOUBLE PRECISION for vdTan, vmdTan	C: Pointer to an array that contains the output vector y .
	COMPLEX for vctan, vmctan	
	DOUBLE COMPLEX for vzTan, vmzTan	
	Fortran 90: REAL, INTENT(OUT) for vstan, vmstan	
	DOUBLE PRECISION, INTENT(OUT) for vdTan, vmdTan	
	COMPLEX, INTENT(OUT) for vctan, vmctan	
	DOUBLE COMPLEX, INTENT(OUT) for vzTan, vmzTan	
	C: float* for vsTan, vmsTan	
	double* for vdTan, vmdTan	
	MKL_Complex8* for vcTan, vmcTan	
	MKL_Complex16* for vzTan, vmzTan	

Description

The v?Tan function computes tangent of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VML provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VML High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VML Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?Tan(x)

Argument	Result	VML Error Status	Exception
+0	+0		
-0	-0		
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Tan}(z) = -i \cdot \text{Tanh}(i \cdot z).$$

v?Acos

Computes inverse cosine of vector elements.

Syntax

Fortran:

```
call vsacos( n, a, y )
call vmsacos( n, a, y, mode )
call vdacos( n, a, y )
```

```

call vmdacos( n, a, y, mode )
call vcacos( n, a, y )
call vmcacos( n, a, y, mode )
call vzacos( n, a, y )
call vmzacos( n, a, y, mode )

```

C:

```

vsAcos( n, a, y );
vmsAcos( n, a, y, mode );
vdAcos( n, a, y );
vmdAcos( n, a, y, mode );
vcAcos( n, a, y );
vmcAcos( n, a, y, mode );
vzAcos( n, a, y );
vmzAcos( n, a, y, mode );

```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const int	Specifies the number of elements to be calculated.
a	FORTRAN 77: REAL for vsacos, vmsacos DOUBLE PRECISION for vdacos, vmdacos COMPLEX for vcacos, vmcacos DOUBLE COMPLEX for vzacos, vmzacos Fortran 90: REAL, INTENT(IN) for vsacos, vmsacos DOUBLE PRECISION, INTENT(IN) for vdacos, vmdacos COMPLEX, INTENT(IN) for vcacos, vmcacos DOUBLE COMPLEX, INTENT(IN) for vzacos, vmzacos	Fortran: Array that specifies the input vector a. C: Pointer to an array that contains the input vector a.

Name	Type	Description
	C: const float* for vsAcos, vmsAcos const double* for vdAcos, vmdAcos const MKL_Complex8* for vcAcos, vmcAcos const MKL_Complex16* for vzAcos, vmzAcos	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vsacos, vmsacos DOUBLE PRECISION for vdacos, vmdacos COMPLEX for vcacos, vmcacos DOUBLE COMPLEX for vzacos, vmzacos Fortran 90: REAL, INTENT (OUT) for vsacos, vmsacos DOUBLE PRECISION, INTENT (OUT) for vdacos, vmdacos COMPLEX, INTENT (OUT) for vcacos, vmcacos DOUBLE COMPLEX, INTENT (OUT) for vzacos, vmzacos	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Acos` function computes inverse cosine of vector elements.

Special Values for Real Function `v?Acos(x)`

Argument	Result	VML Error Status	Exception
+0	$+\pi/2$		

Argument	Result	VML Error Status	Exception
-0	+π/2		
+1	+0		
-1	+π		
X > 1	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function v?Acos(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	$+\frac{3\pi}{4} - i \cdot \infty$	$+\frac{\pi}{2} - i \cdot \infty$	$+\frac{\pi}{2} - i \cdot \infty$	$+\frac{\pi}{2} - i \cdot \infty$	$+\frac{\pi}{2} - i \cdot \infty$	$+\frac{\pi}{4} - i \cdot \infty$	QNAN-i·∞
+i·Y	$+\pi - i \cdot \infty$					$+0 - i \cdot \infty$	QNAN+i·QNAN
+i·0	$+\pi - i \cdot \infty$		$+\frac{\pi}{2} - i \cdot 0$	$+\frac{\pi}{2} - i \cdot 0$		$+0 - i \cdot \infty$	QNAN+i·QNAN
-i·0	$+\pi + i \cdot \infty$		$+\frac{\pi}{2} + i \cdot \infty$	$+\frac{\pi}{2} + i \cdot \infty$		$+0 + i \cdot \infty$	QNAN+i·QNAN
-i·Y	$+\pi + i \cdot \infty$					$+0 + i \cdot \infty$	QNAN+i·QNAN
-i·∞	$+\frac{3\pi}{4} + i \cdot \infty$	$+\frac{\pi}{2} + i \cdot \infty$	$+\frac{\pi}{2} + i \cdot \infty$	$+\frac{\pi}{2} + i \cdot \infty$	$+\frac{\pi}{2} + i \cdot \infty$	$+\frac{\pi}{4} + i \cdot \infty$	QNAN+i·∞
+i·NAN	QNAN+i·∞	QNAN+i·QNAN	$+\frac{\pi}{2} + i \cdot \text{QNAN}$	$+\frac{\pi}{2} + i \cdot \text{QNAN}$	QNAN+i·QNAN	QNAN+i·∞	QNAN+i·QNAN

Notes:

- raises INVALID exception when real or imaginary part of the argument is SNAN
- $\text{Acos}(\text{CONJ}(z)) = \text{CONJ}(\text{Acos}(z))$.

v?Asin

Computes inverse sine of vector elements.

Syntax

Fortran:

```
call vsasin( n, a, y )
call vmsasin( n, a, y, mode )
call vdasin( n, a, y )
call vmdasin( n, a, y, mode )
call vcasin( n, a, y )
call vmcasin( n, a, y, mode )
call vzasin( n, a, y )
call vmzasin( n, a, y, mode )
```

C:

```
vsAsin( n, a, y );
vmsAsin( n, a, y, mode );
vdAsin( n, a, y );
vmdAsin( n, a, y, mode );
vcAsin( n, a, y );
vmcAsin( n, a, y, mode );
vzAsin( n, a, y );
vmzAsin( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
a	FORTRAN 77: REAL for vsasin, vmsasin DOUBLE PRECISION for vdasin, vmdasin COMPLEX for vcasin, vmcasin DOUBLE COMPLEX for vzasin, vmzasin Fortran 90: REAL, INTENT(IN) for vsasin, vmsasin DOUBLE PRECISION, INTENT(IN) for vdasin, vmdasin COMPLEX, INTENT(IN) for vcasin, vmcasin DOUBLE COMPLEX, INTENT(IN) for vzasin, vmzasin C: const float* for vsAsin, vmsAsin const double* for vdAsin, vmdAsin const MKL_Complex8* for vcAsin, vmcAsin const MKL_Complex16* for vzAsin, vmzAsin	Fortran: Array that specifies the input vector a. C: Pointer to an array that contains the input vector a.

Name	Type	Description
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vsasin, vmsasin DOUBLE PRECISION for vdasin, vmdasin COMPLEX for vcasin, vmcasin DOUBLE COMPLEX for vzasin, vmzasin Fortran 90: REAL, INTENT(OUT) for vsasin, vmsasin DOUBLE PRECISION, INTENT(OUT) for vdasin, vmdasin COMPLEX, INTENT(OUT) for vcasin, vmcasin DOUBLE COMPLEX, INTENT(OUT) for vzasin, vmzasin C: float* for vsAsin, vmsAsin double* for vdAsin, vmdAsin MKL_Complex8* for vcAsin, vmcAsin MKL_Complex16* for vzAsin, vmzAsin	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Asin` function computes inverse sine of vector elements.

Special Values for Real Function `v?Asin(x)`

Argument	Result	VML Error Status	Exception
+0	+0		
-0	-0		
+1	+π/2		
-1	-π/2		
X > 1	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

```
Asin(z) = -i*Asinh(i*z).
```

v?Atan

Computes inverse tangent of vector elements.

Syntax

Fortran:

```
call vsatan( n, a, y )
call vmsatan( n, a, y, mode )
call vdatan( n, a, y )
call vmdatan( n, a, y, mode )
call vcatan( n, a, y )
call vmcatan( n, a, y, mode )
call vzatan( n, a, y )
call vmzatan( n, a, y, mode )
```

C:

```
vsAtan( n, a, y );
vmsAtan( n, a, y, mode );
vdAtan( n, a, y );
vmdAtan( n, a, y, mode );
vcAtan( n, a, y );
vmcAtan( n, a, y, mode );
vzAtan( n, a, y );
vmzAtan( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
a	FORTTRAN 77: REAL for vsatan, vmsatan DOUBLE PRECISION for vdatan, vmdatan COMPLEX for vcatan, vmcatan	Fortran: Array that specifies the input vector a. C: Pointer to an array that contains the input vector a.

Name	Type	Description
	DOUBLE COMPLEX for vzatan, vmzatan	
	Fortran 90: REAL, INTENT(IN) for vsatan, vmsatan	
	DOUBLE PRECISION, INTENT(IN) for vdatan, vmdatan	
	COMPLEX, INTENT(IN) for vcatan, vmcatan	
	DOUBLE COMPLEX, INTENT(IN) for vzatan, vmzatan	
	C: const float* for vsAtan, vmsAtan	
	const double* for vdAsin, vmdAtan	
	const MKL_Complex8* for vcAtan, vmcAtan	
	const MKL_Complex16* for vzAsin, vmzAtan	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN)	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.
	C: const MKL_INT64	

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vsatan, vmsatan	Fortran: Array that specifies the output vector y.
	DOUBLE PRECISION for vdatan, vmdatan	C: Pointer to an array that contains the output vector y.
	COMPLEX for vcatan, vmcatan	
	DOUBLE COMPLEX for vzatan, vmzatan	
	Fortran 90: REAL, INTENT(OUT) for vsatan, vmsatan	
	DOUBLE PRECISION, INTENT(OUT) for vdatan, vmdatan	
	COMPLEX, INTENT(OUT) for vcatan, vmcatan	
	DOUBLE COMPLEX, INTENT(OUT) for vzatan, vmzatan	
	C: float* for vsAtan, vmsAtan	

Name	Type	Description
	double* for vdAsin, vmdAtan	
	MKL_Complex8* for vcAtan, vmcAtan	
	MKL_Complex16* for vzAsin, vmzAtan	

Description

The v?Atan function computes inverse tangent of vector elements.

Special Values for Real Function v?Atan(x)

Argument	Result	VML Error Status	Exception
+0	+0		
-0	-0		
+∞	+π/2		
-∞	-π/2		
QNAN	QNAN		
SNAN	QNAN		INVALID

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Atan}(z) = -i \cdot \text{Atanh}(i \cdot z).$$

v?Atan2

Computes four-quadrant inverse tangent of elements of two vectors.

Syntax

Fortran:

```
call vsatan2( n, a, b, y )
call vmsatan2( n, a, b, y, mode )
call vdatan2( n, a, b, y )
call vmdatn2( n, a, b, y, mode )
```

C:

```
vsAtan2( n, a, b, y );
vmsAtan2( n, a, b, y, mode );
vdAtan2( n, a, b, y );
vmdatn2( n, a, b, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a, b</i>	FORTRAN 77: REAL for vsatan2, vmsatan2 DOUBLE PRECISION for vdatan2, vmdataan2 Fortran 90: REAL, INTENT(IN) for vsatan2, vmsatan2 DOUBLE PRECISION, INTENT(IN) for vdatan2, vmdataan2 C: const float* for vsAtan2, vmsAtan2 const double* for vdAtan2, vmdAtan2	Fortran: Arrays that specify the input vectors <i>a</i> and <i>b</i> . C: Pointers to arrays that contain the input vectors <i>a</i> and <i>b</i> .
<i>mode</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vsatan2, vmsatan2 DOUBLE PRECISION for vdatan2, vmdataan2 Fortran 90: REAL, INTENT(OUT) for vsatan2, vmsatan2 DOUBLE PRECISION, INTENT(OUT) for vdatan2, vmdataan2 C: float* for vsAtan2, vmsAtan2 double* for vdAtan2, vmdAtan2	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Atan2` function computes four-quadrant inverse tangent of elements of two vectors.

The elements of the output vector *y* are computed as the four-quadrant arctangent of $a[i] / b[i]$.

Special values for Real Function `v?Atan2(x)`

Argument 1	Argument 2	Result	Exception
$-\infty$	$-\infty$	$-3\pi/4$	

Argument 1	Argument 2	Result	Exception
-∞	X < +0	-π/2	
-∞	-0	-π/2	
-∞	+0	-π/2	
-∞	X > +0	-π/2	
-∞	+∞	-π/4	
X < +0	-∞	-π	
X < +0	-0	-π/2	
X < +0	+0	-π/2	
X < +0	+∞	-0	
-0	-∞	-π	
-0	X < +0	-π	
-0	-0	-π	
-0	+0	-0	
-0	X > +0	-0	
-0	+∞	-0	
+0	-∞	+π	
+0	X < +0	+π	
+0	-0	+π	
+0	+0	+0	
+0	X > +0	+0	
+0	+∞	+0	
X > +0	-∞	+π	
X > +0	-0	+π/2	
X > +0	+0	+π/2	
X > +0	+∞	+0	
+∞	-∞	-3*π/4	
+∞	X < +0	+π/2	
+∞	-0	+π/2	
+∞	+0	+π/2	
+∞	X > +0	+π/2	
+∞	+∞	+π/4	
X > +0	QNAN	QNAN	
X > +0	SNAN	QNAN	INVALID
QNAN	X > +0	QNAN	
SNAN	X > +0	QNAN	INVALID
QNAN	QNAN	QNAN	
QNAN	SNAN	QNAN	INVALID
SNAN	QNAN	QNAN	INVALID
SNAN	SNAN	QNAN	INVALID

Hyperbolic Functions

v?Cosh

Computes hyperbolic cosine of vector elements.

Syntax

Fortran:

```
call vscosh( n, a, y )
call vmscosh( n, a, y, mode )
call vdcosh( n, a, y )
call vmdcosh( n, a, y, mode )
call vccosh( n, a, y )
call vmccosh( n, a, y, mode )
call vzcosh( n, a, y )
call vmzcosh( n, a, y, mode )
```

C:

```
vsCosh( n, a, y );
vmsCosh( n, a, y, mode );
vdCosh( n, a, y );
vmdCosh( n, a, y, mode );
vcCosh( n, a, y );
vmcCosh( n, a, y, mode );
vzCosh( n, a, y );
vmzCosh( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
a	FORTRAN 77: REAL for vscosh, vmscosh DOUBLE PRECISION for vdcosh, vmdcosh COMPLEX for vccosh, vmccosh DOUBLE COMPLEX for vzcosh, vmzcosh Fortran 90: REAL, INTENT(IN) for vscosh, vmscosh	Fortran: Array that specifies the input vector a. C: Pointer to an array that contains the input vector a.

Name	Type	Description
	DOUBLE PRECISION, INTENT(IN) for vdcoh, vmdcoh	
	COMPLEX, INTENT(IN) for vcccoh, vmcccoh	
	DOUBLE COMPLEX, INTENT(IN) for vzccoh, vmzcoh	
	C: const float* for vsCosh, vmsCosh	
	const double* for vdCosh, vmdCosh	
	const MKL_Complex8* for vcCosh, vmCosh	
	const MKL_Complex16* for vzCosh, vmzCosh	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Cosh Function

Data Type	Threshold Limitations on Input Parameters
single precision	-Ln(FLT_MAX)-Ln2 < a[i] < Ln(FLT_MAX)+Ln2
double precision	-Ln(DBL_MAX)-Ln2 < a[i] < Ln(DBL_MAX)+Ln2

Precision overflow thresholds for the complex v?Cosh function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vscosh, vmscosh	Fortran: Array that specifies the output vector y.
	DOUBLE PRECISION for vdcoh, vmdcoh	C: Pointer to an array that contains the output vector y.
	COMPLEX for vcccoh, vmcccoh	
	DOUBLE COMPLEX for vzccoh, vmzcoh	
	Fortran 90: REAL, INTENT(OUT) for vscosh, vmscosh	
	DOUBLE PRECISION, INTENT(OUT) for vdcoh, vmdcoh	
	COMPLEX, INTENT(OUT) for vcccoh, vmcccoh	
	DOUBLE COMPLEX, INTENT(OUT) for vzccoh, vmzcoh	

Name	Type	Description
	C: float* for vsCosh, vmsCosh	
	double* for vdCosh, vmdCosh	
	MKL_Complex8* for vcCosh, vmCosh	
	MKL_Complex16* for vzCosh, vmzCosh	

Description

The $v?Cosh$ function computes hyperbolic cosine of vector elements.

Special Values for Real Function $v?Cosh(x)$

Argument	Result	VML Error Status	Exception
+0	+1		
-0	+1		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
X < -overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
+∞	+∞		
-∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function $v?Cosh(z)$

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+∞+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN-i·0 INVALID	QNAN+i·0 INVALID	QNAN+i·QNAN INVALID	+∞+i·QNAN INVALID	QNAN+i·QNAN
+i·Y	+∞·Cos(Y)- i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
+i·0	+∞-i·0		+1-i·0	+1+i·0		+∞+i·0	QNAN+i·0
-i·0	+∞+i·0		+1+i·0	+1-i·0		+∞-i·0	QNAN-i·0
-i·Y	+∞·Cos(Y)- i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
-i·∞	+∞+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·0 INVALID	QNAN-i·0 INVALID	QNAN+i·QNAN INVALID	+∞+i·QNAN INVALID	QNAN+i·QNAN
+i·NAN	+∞+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN-i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Notes:

- raises the INVALID exception when the real or imaginary part of the argument is SNAN
- raises the OVERFLOW exception and sets the VML Error Status to VML_STATUS_OVERFLOW in the case of overflow, that is, when $RE(z)$, $IM(z)$ are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- $\text{Cosh}(\text{CONJ}(z)) = \text{CONJ}(\text{Cosh}(z))$
- $\text{Cosh}(-z) = \text{Cosh}(z)$.

v?Sinh

Computes hyperbolic sine of vector elements.

Syntax

Fortran:

```
call vssinh( n, a, y )
call vmssinh( n, a, y, mode )
call vdsinh( n, a, y )
call vmdsinh( n, a, y, mode )
call vcsinh( n, a, y )
call vmcsinh( n, a, y, mode )
call vzsinh( n, a, y )
call vmzsinh( n, a, y, mode )
```

C:

```
vsSinh( n, a, y );
vmsSinh( n, a, y, mode );
vdSinh( n, a, y );
vmdSinh( n, a, y, mode );
vcSinh( n, a, y );
vmcSinh( n, a, y, mode );
vzSinh( n, a, y );
vmzSinh( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
a	FORTRAN 77: REAL for vssinh, vmssinh DOUBLE PRECISION for vdsinh, vmdsinh COMPLEX for vcsinh, vmcsinh DOUBLE COMPLEX for vzsinh, vmzsinh	Fortran: Array that specifies the input vector a. C: Pointer to an array that contains the input vector a.

Name	Type	Description
	<pre> Fortran 90: REAL, INTENT(IN) for vssinh, vmssinh DOUBLE PRECISION, INTENT(IN) for vdsinh, vmdsinh COMPLEX, INTENT(IN) for vcsinh, vmcsinh DOUBLE COMPLEX, INTENT(IN) for vzsinh, vmzsinh C: const float* for vsSinh, vmsSinh const double* for vdSinh, vmdSinh const MKL_Complex8* for vcSinh, vmcSinh const MKL_Complex16* for vzSinh, vmzSinh </pre>	
mode	<pre> FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64 </pre>	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Sinh Function

Data Type	Threshold Limitations on Input Parameters
single precision	$-\ln(\text{FLT_MAX}) - \ln 2 < a[i] < \ln(\text{FLT_MAX}) + \ln 2$
double precision	$-\ln(\text{DBL_MAX}) - \ln 2 < a[i] < \ln(\text{DBL_MAX}) + \ln 2$

Precision overflow thresholds for the complex v?Sinh function are beyond the scope of this document.

Output Parameters

Name	Type	Description
y	<pre> FORTRAN 77: REAL for vssinh, vmssinh DOUBLE PRECISION for vdsinh, vmdsinh COMPLEX for vcsinh, vmcsinh DOUBLE COMPLEX for vzsinh, vmzsinh Fortran 90: REAL, INTENT(OUT) for vssinh, vmssinh DOUBLE PRECISION, INTENT(OUT) for vdsinh, vmdsinh COMPLEX, INTENT(OUT) for vcsinh, vmcsinh </pre>	<p>Fortran: Array that specifies the output vector <i>y</i>.</p> <p>C: Pointer to an array that contains the output vector <i>y</i>.</p>

Name	Type	Description
		<pre>DOUBLE COMPLEX, INTENT(OUT) for vzsinh, vmzsinh C: float* for vsSinh, vmsSinh double* for vdSinh, vmdSinh MKL_Complex8* for vcSinh, vmcSinh MKL_Complex16* for vzSinh, vmzSinh</pre>

Description

The $v?Sinh$ function computes hyperbolic sine of vector elements.

Special Values for Real Function $v?Sinh(x)$

Argument	Result	VML Error Status	Exception
+0	+0		
-0	-0		
X > overflow	+∞	VML_STATUS_OVERFLOW	OVERFLOW
X < -overflow	-∞	VML_STATUS_OVERFLOW	OVERFLOW
+∞	+∞		
-∞	-∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function $v?Sinh(z)$

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-∞+i·QNAN INVALID	QNAN+i·QNAN INVALID	-0+i·QNAN INVALID	+0+i·QNAN INVALID	QNAN+i·QNAN INVALID	+∞+i·QNAN INVALID	QNAN+i·QNAN
+i·Y	-∞·Cos(Y)+ i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
+i·0	-∞+i·0		-0+i·0	+0+i·0		+∞+i·0	QNAN+i·0
-i·0	-∞-i·0		-0-i·0	+0-i·0		+∞-i·0	QNAN-i·0
-i·Y	-∞·Cos(Y)+ i·∞·Sin(Y)					+∞·CIS(Y)	QNAN+i·QNAN
-i·∞	-∞+i·QNAN INVALID	QNAN+i·QNAN INVALID	-0+i·QNAN INVALID	+0+i·QNAN INVALID	QNAN+i·QNAN INVALID	+∞+i·QNAN INVALID	QNAN+i·QNAN
+i·NAN	-∞+i·QNAN	QNAN+i·QNAN	-0+i·QNAN	+0+i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Notes:

- raises the `INVALID` exception when the real or imaginary part of the argument is `SNAN`
- raises the `OVERFLOW` exception and sets the VML Error Status to `VML_STATUS_OVERFLOW` in the case of overflow, that is, when `RE(z)`, `IM(z)` are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- $\text{Sinh}(\text{CONJ}(z)) = \text{CONJ}(\text{Sinh}(z))$

- $\text{Sinh}(-z) = -\text{Sinh}(z)$.

v?Tanh

Computes hyperbolic tangent of vector elements.

Syntax

Fortran:

```
call vstanh( n, a, y )
call vmstanh( n, a, y, mode )
call vdtanh( n, a, y )
call vmdtanh( n, a, y, mode )
call vctanh( n, a, y )
call vmctanh( n, a, y, mode )
call vztanh( n, a, y )
call vmztanh( n, a, y, mode )
```

C:

```
vsTanh( n, a, y );
vmsTanh( n, a, y, mode );
vdTanh( n, a, y );
vmdTanh( n, a, y, mode );
vcTanh( n, a, y );
vmcTanh( n, a, y, mode );
vzTanh( n, a, y );
vmzTanh( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
a	FORTAN 77: REAL for vstanh, vmstanh DOUBLE PRECISION for vdtanh, vmdtanh COMPLEX for vctanh, vmctanh	Fortran: Array that specifies the input vector a. C: Pointer to an array that contains the input vector a.

Name	Type	Description
	DOUBLE COMPLEX for vztanh, vmztanh	
	Fortran 90: REAL, INTENT(IN) for vstanh, vmstanh	
	DOUBLE PRECISION, INTENT(IN) for vdtanh, vmdtanh	
	COMPLEX, INTENT(IN) for vctanh, vmctanh	
	DOUBLE COMPLEX, INTENT(IN) for vztanh, vmztanh	
	C: const float* for vsTanh, vmsTanh	
	const double* for vdTanh, vmdTanh	
	const MKL_Complex8* for vcTanh, vmcTanh	
	const MKL_Complex16* for vzTanh, vmzTanh	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vstanh, vmstanh	Fortran: Array that specifies the output vector y.
	DOUBLE PRECISION for vdtanh, vmdtanh	C: Pointer to an array that contains the output vector y.
	COMPLEX for vctanh, vmctanh	
	DOUBLE COMPLEX for vztanh, vmztanh	
	Fortran 90: REAL, INTENT(OUT) for vstanh, vmstanh	
	DOUBLE PRECISION, INTENT(OUT) for vdtanh, vmdtanh	
	COMPLEX, INTENT(OUT) for vctanh, vmctanh	
	DOUBLE COMPLEX, INTENT(OUT) for vztanh, vmztanh	
	C: float* for vsTanh, vmsTanh	

Name	Type	Description
	double* for vdTanh, vmdTanh	
	MKL_Complex8* for vcTanh, vmcTanh	
	MKL_Complex16* for vzTanh, vmzTanh	

Description

The $v\text{Tanh}$ function computes hyperbolic tangent of vector elements.

Special Values for Real Function $v\text{Tanh}(x)$

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	+1	
$-\infty$	-1	
QNAN	QNAN	
SNAN	QNAN	INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function $v\text{Tanh}(z)$

$\text{RE}(z)$ $i \cdot \text{IM}(z)$	$-\infty$	$-x$	-0	$+0$	$+x$	$+\infty$	NAN
$+i \cdot \infty$	$-1+i \cdot 0$	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	$+1+i \cdot 0$	QNAN+i·QNAN
$+i \cdot Y$	$-1+i \cdot 0 \cdot \text{Tan}(Y)$					$+1+i \cdot 0 \cdot \text{Tan}(Y)$	QNAN+i·QNAN
$+i \cdot 0$	$-1+i \cdot 0$		$-0+i \cdot 0$	$+0+i \cdot 0$		$+1+i \cdot 0$	QNAN+i·0
$-i \cdot 0$	$-1-i \cdot 0$		$-0-i \cdot 0$	$+0-i \cdot 0$		$+1-i \cdot 0$	QNAN-i·0
$-i \cdot Y$	$-1+i \cdot 0 \cdot \text{Tan}(Y)$					$+1+i \cdot 0 \cdot \text{Tan}(Y)$	QNAN+i·QNAN
$-i \cdot \infty$	$-1-i \cdot 0$	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	QNAN+i·QNAN INVALID	$+1-i \cdot 0$	QNAN+i·QNAN
$+i \cdot \text{NAN}$	$-1+i \cdot 0$	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	$+1+i \cdot 0$	QNAN+i·QNAN

Notes:

- raises INVALID exception when real or imaginary part of the argument is SNAN
- $\text{Tanh}(\text{CONJ}(z)) = \text{CONJ}(\text{Tanh}(z))$
- $\text{Tanh}(-z) = -\text{Tanh}(z)$.

v?Acosh

Computes inverse hyperbolic cosine (nonnegative) of vector elements.

Syntax

Fortran:

```
call vsacosh( n, a, y )
call vmsacosh( n, a, y, mode )
```

```

call vdacosh( n, a, y )
call vmdacosh( n, a, y, mode )
call vcacosh( n, a, y )
call vmcacosh( n, a, y, mode )
call vzacosh( n, a, y )
call vmzacosh( n, a, y, mode )

```

C:

```

vsAcosh( n, a, y );
vmsAcosh( n, a, y, mode );
vdAcosh( n, a, y );
vmdAcosh( n, a, y, mode );
vcAcosh( n, a, y );
vmcAcosh( n, a, y, mode );
vzAcosh( n, a, y );
vmzAcosh( n, a, y, mode );

```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER, INTENT(IN) <small>C:</small> const MKL_INT	Specifies the number of elements to be calculated.
a	<small>FORTRAN 77:</small> REAL for vsacosh, <small>vmsacosh</small> <small>DOUBLE PRECISION</small> for vdacosh, <small>vmdacosh</small> <small>COMPLEX</small> for vcacosh, vmcacosh <small>DOUBLE COMPLEX</small> for vzacosh, <small>vmzacosh</small> <small>Fortran 90:</small> REAL, INTENT(IN) for <small>vsacosh, vmsacosh</small> <small>DOUBLE PRECISION, INTENT(IN)</small> for <small>vdacosh, vmdacosh</small> <small>COMPLEX, INTENT(IN)</small> for vcacosh, <small>vmcacosh</small> <small>DOUBLE COMPLEX, INTENT(IN)</small> for <small>vzacosh, vmzacosh</small>	<small>Fortran:</small> Array that specifies the input vector <i>a</i> . <small>C:</small> Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	<pre>C: const float* for vsAcosh, vmsAcosh const double* for vdAcosh, vmdAcosh const MKL_Complex8* for vcAcosh, vmcAcosh const MKL_Complex16* for vzAcosh, vmzAcosh</pre>	
mode	<pre>FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64</pre>	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
Y	<pre>FORTRAN 77: REAL for vsacosh, vmsacosh DOUBLE PRECISION for vdacosh, vmdacosh COMPLEX for vcacosh, vmcacosh DOUBLE COMPLEX for vzacosh, vmzacosh Fortran 90: REAL, INTENT(OUT) for vsacosh, vmsacosh DOUBLE PRECISION, INTENT(OUT) for vdacosh, vmdacosh COMPLEX, INTENT(OUT) for vcacosh, vmcacosh DOUBLE COMPLEX, INTENT(OUT) for vzacosh, vmzacosh C: float* for vsAcosh, vmsAcosh double* for vdAcosh, vmdAcosh MKL_Complex8* for vcAcosh, vmcAcosh MKL_Complex16* for vzAcosh, vmzAcosh</pre>	<p>Fortran: Array that specifies the output vector y.</p> <p>C: Pointer to an array that contains the output vector y.</p>

Description

The `v?Acosh` function computes inverse hyperbolic cosine (nonnegative) of vector elements.

Special Values for Real Function v?Acosh(x)

Argument	Result	VML Error Status	Exception
+1	+0		
X < +1	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	+∞		
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function v?Acosh(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	+∞ + i · $\frac{3\pi}{4}$	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	+∞+i·π					+∞+i·0	QNAN+i·QNAN
+i·0	+∞+i·π		+0+i·π/2	+0+i·π/2		+∞+i·0	QNAN+i·QNAN
-i·0	+∞+i·π		+0+i·π/2	+0+i·π/2		+∞+i·0	QNAN+i·QNAN
-i·Y	+∞+i·π					+∞+i·0	QNAN+i·QNAN
-i·∞	+∞ - i · $\frac{3\pi}{4}$	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	+∞+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	QNAN+i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Notes:

- raises INVALID exception when real or imaginary part of the argument is SNAN
- $\text{Acosh}(\text{CONJ}(z)) = \text{CONJ}(\text{Acosh}(z))$.

v?Asinh

Computes inverse hyperbolic sine of vector elements.

Syntax**Fortran:**

```
call vsasinh( n, a, y )
call vmsasinh( n, a, y, mode )
call vdasin( n, a, y )
call vmdasin( n, a, y, mode )
call vcasin( n, a, y )
call vmcasin( n, a, y, mode )
call vzasin( n, a, y )
call vmzasin( n, a, y, mode )
```

C:

```
vsAsinh( n, a, y );
```

```
vmsAsinh( n, a, y, mode );
vdAsinh( n, a, y );
vmdAsinh( n, a, y, mode );
vcAsinh( n, a, y );
vmcAsinh( n, a, y, mode );
vzAsinh( n, a, y );
vmzAsinh( n, a, y, mode );
```

Include Files

- **Fortran:** mkl_vml.f90
- **C:** mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77: REAL for vsasinh, vmsasinh DOUBLE PRECISION for vdasinh, vmdasinh COMPLEX for vcasinh, vmcasinh DOUBLE COMPLEX for vzasinh, vmzasinh Fortran 90: REAL, INTENT(IN) for vsasinh, vmsasinh DOUBLE PRECISION, INTENT(IN) for vdasinh, vmdasinh COMPLEX, INTENT(IN) for vcasinh, vmcasinh DOUBLE COMPLEX, INTENT(IN) for vzasinh, vmzasinh C: const float* for vsAinh, vmsAinh const double* for vdAinh, vmdAinh const MKL_Complex8* for vcAinh, vmcAinh const MKL_Complex16* for vzAinh, vmzAinh	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
mode	<p>FORTRAN 77: INTEGER*8</p> <p>Fortran 90: INTEGER(KIND=8), INTENT(IN)</p> <p>C: const MKL_INT64</p>	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<p>FORTRAN 77: REAL for vsasinh, vmsasinh</p> <p>DOUBLE PRECISION for vdasin, vmdasin</p> <p>COMPLEX for vcasin, vmcasin</p> <p>DOUBLE COMPLEX for vzasin, vmzasin</p> <p>Fortran 90: REAL, INTENT(OUT) for vsasinh, vmsasinh</p> <p>DOUBLE PRECISION, INTENT(OUT) for vdasin, vmdasin</p> <p>COMPLEX, INTENT(OUT) for vcasin, vmcasin</p> <p>DOUBLE COMPLEX, INTENT(OUT) for vzasin, vmzasin</p> <p>C: float* for vsAsinh, vmsAsinh</p> <p>double* for vdAsinh, vmdAsinh</p> <p>MKL_Complex8* for vcAsinh, vmcasinh</p> <p>MKL_Complex16* for vzAsinh, vmzasinh</p>	<p>Fortran: Array that specifies the output vector <i>y</i>.</p> <p>C: Pointer to an array that contains the output vector <i>y</i>.</p>

Description

The `v?Asinh` function computes inverse hyperbolic sine of vector elements.

Special Values for Real Function `v?Asinh(x)`

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
QNAN	QNAN	
SNAN	QNAN	INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function v?Asinh(z)

RE(z) i·IM(z)	-∞	-X	-0	+0	+X	+∞	NAN
+i·∞	-∞+i·π/4	-∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/2	+∞+i·π/4	+∞+i·QNAN
+i·Y	-∞+i·0					+∞+i·0	QNAN+i·QNAN
+i·0	+∞+i·0		+0+i·0	+0+i·0		+∞+i·0	QNAN+i·QNAN
-i·0	-∞-i·0		-0-i·0	+0-i·0		+∞-i·0	QNAN-i·QNAN
-i·Y	-∞-i·0					+∞-i·0	QNAN+i·QNAN
-i·∞	-∞-i·π/4	-∞-i·π/2	-∞-i·π/2	+∞-i·π/2	+∞-i·π/2	+∞-i·π/4	+∞+i·QNAN
+i·NAN	-∞+i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	QNAN +i·QNAN	+∞+i·QNAN	QNAN+i·QNAN

Notes:

- raises INVALID exception when real or imaginary part of the argument is SNAN
- Asinh(CONJ(z))=CONJ(Asinh(z))
- Asinh(-z)=-Asinh(z).

v?Atanh

Computes inverse hyperbolic tangent of vector elements.

Syntax**Fortran:**

```
call vsatanh( n, a, y )
call vmsatanh( n, a, y, mode )
call vdatanh( n, a, y )
call vmdatanh( n, a, y, mode )
call vcataanh( n, a, y )
call vmcatanh( n, a, y, mode )
call vzatanh( n, a, y )
call vmzataanh( n, a, y, mode )
```

C:

```
vsAtanh( n, a, y );
vmsAtanh( n, a, y, mode );
vdAtanh( n, a, y );
vmdAtanh( n, a, y, mode );
vcAtanh( n, a, y );
vmcAtanh( n, a, y, mode );
vzAtanh( n, a, y );
vmzAtanh( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER, INTENT(IN)</p> <p>C: const MKL_INT</p>	Specifies the number of elements to be calculated.
<i>a</i>	<p>FORTRAN 77: REAL for vsatanh, vmsatanh</p> <p>DOUBLE PRECISION for vdatanh, vmdataanh</p> <p>COMPLEX for vcataanh, vmcatanh</p> <p>DOUBLE COMPLEX for vzatanh, vmzatanh</p> <p>Fortran 90: REAL, INTENT(IN) for vsatanh, vmsatanh</p> <p>DOUBLE PRECISION, INTENT(IN) for vdatanh, vmdataanh</p> <p>COMPLEX, INTENT(IN) for vcataanh, vmcatanh</p> <p>DOUBLE COMPLEX, INTENT(IN) for vzatanh, vmzatanh</p> <p>C: const float* for vsAtanh, vmsAtanh</p> <p>const double* for vdAtanh, vmdAtanh</p> <p>const MKL_Complex8* for vcAtanh, vmcAtanh</p> <p>const MKL_Complex16* for vzAtanh, vmzAtanh</p>	<p>Fortran: Array that specifies the input vector <i>a</i>.</p> <p>C: Pointer to an array that contains the input vector <i>a</i>.</p>
<i>mode</i>	<p>FORTRAN 77: INTEGER*8</p> <p>Fortran 90: INTEGER(KIND=8), INTENT(IN)</p> <p>C: const MKL_INT64</p>	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vsatanh, vmsatanh	Fortran: Array that specifies the output vector <i>y</i> .

Name	Type	Description
	DOUBLE PRECISION for vdatanh, vmdataanh	C: Pointer to an array that contains the output vector y .
	COMPLEX for vcatanh, vmcatanh	
	DOUBLE COMPLEX for vzatanh, vmzatanh	
	Fortran 90: REAL, INTENT(OUT) for vsatanh, vmsatanh	
	DOUBLE PRECISION, INTENT(OUT) for vdatanh, vmdataanh	
	COMPLEX, INTENT(OUT) for vcatanh, vmcatanh	
	DOUBLE COMPLEX, INTENT(OUT) for vzatanh, vmzatanh	
	C: float* for vsAtanh, vmsAtanh double* for vdAtanh, vmdAtanh	
	MKL_Complex8* for vcAtanh, vmcAtanh	
	MKL_Complex16* for vzAtanh, vmzAtanh	

Description

The $v?Atanh$ function computes inverse hyperbolic tangent of vector elements.

Special Values for Real Function $v?Atanh(x)$

Argument	Result	VML Error Status	Exception
+1	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-1	$-\infty$	VML_STATUS_SING	ZERODIVIDE
$ X > 1$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See the [Special Value Notations](#) section for the conventions used in the table below.

Special Values for Complex Function $v?Atanh(z)$

$RE(z)$ $i \cdot IM(z)$	$-\infty$	$-x$	-0	$+0$	$+x$	$+\infty$	NAN
$+i \cdot \infty$	$-0+i\pi/2$	$-0+i\pi/2$	$-0+i\pi/2$	$+0+i\pi/2$	$+0+i\pi/2$	$+0+i\pi/2$	$+0+i\pi/2$
$+i \cdot Y$	$-0+i\pi/2$					$+0+i\pi/2$	$QNAN+iQNAN$
$+i \cdot 0$	$-0+i\pi/2$		$-0+i \cdot 0$	$+0+i \cdot 0$		$+0+i\pi/2$	$QNAN+iQNAN$
$-i \cdot 0$	$-0-i\pi/2$		$-0-i \cdot 0$	$+0-i \cdot 0$		$+0-i\pi/2$	$QNAN-iQNAN$

RE(z)	-∞	-X	-0	+0	+X	+∞	NAN
i·IM(z)							
-i·Y	-0-i·π/2					+0-i·π/2	QNAN+i·QNAN
-i·∞	-0-i·π/2	-0-i·π/2	-0-i·π/2	+0-i·π/2	+0-i·π/2	+0-i·π/2	+0-i·π/2
+i·NAN	-0+i·QNAN +i·QNAN	QNAN +i·QNAN	-0+i·QNAN	+0+i·QNAN	QNAN +i·QNAN	+0+i·QNAN	QNAN+i·QNAN

Notes:

- Atanh (+-1+-i*0) = +-∞+-i*0, and ZERO_DIVIDE exception is raised
- raises INVALID exception when real or imaginary part of the argument is SNAN
- Atanh (CONJ(z)) = CONJ(Atanh(z))
- Atanh(-z) = -Atanh(z).

Special Functions

v?Erf

Computes the error function value of vector elements.

Syntax

Fortran:

```
call vserf( n, a, y )
call vmserf( n, a, y, mode )
call vdervf( n, a, y )
call vmdervf( n, a, y, mode )
```

C:

```
vsErf( n, a, y );
vmsErf( n, a, y, mode );
vdErf( n, a, y );
vmdErf( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.

Name	Type	Description
a	<p>FORTRAN 77: REAL for vserf, vmserf</p> <p>DOUBLE PRECISION for vderf, vmderf</p> <p>Fortran 90: REAL, INTENT(IN) for vserf, vmserf</p> <p>DOUBLE PRECISION, INTENT(IN) for vderf, vmderf</p> <p>C: const float* for vsErf, vmsErf const double* for vdErf, vmdErf</p>	<p>Fortran: Array, specifies the input vector <i>a</i>.</p> <p>C: Pointer to an array that contains the input vector <i>a</i>.</p>
mode	<p>FORTRAN 77: INTEGER*8</p> <p>Fortran 90: INTEGER(KIND=8), INTENT(IN)</p> <p>C: const MKL_INT64</p>	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	<p>FORTRAN 77: REAL for vserf, vmserf</p> <p>DOUBLE PRECISION for vderf, vmderf</p> <p>Fortran 90: REAL, INTENT(OUT) for vserf, vmserf</p> <p>DOUBLE PRECISION, INTENT(OUT) for vderf, vmderf</p> <p>C: float* for vsErf, vmsErf double* for vdErf, vmdErf</p>	<p>Fortran: Array, specifies the output vector <i>y</i>.</p> <p>C: Pointer to an array that contains the output vector <i>y</i>.</p>

Description

The `Erf` function computes the error function values for elements of the input vector *a* and writes them to the output vector *y*.

The error function is defined as given by:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Useful relations:

1. $\text{erfc}(x) = 1 - \text{erf}(x) ,$

where erfc is the complementary error function.

$$2. \Phi(x) = \frac{1}{2} \text{erf}(x/\sqrt{2}),$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

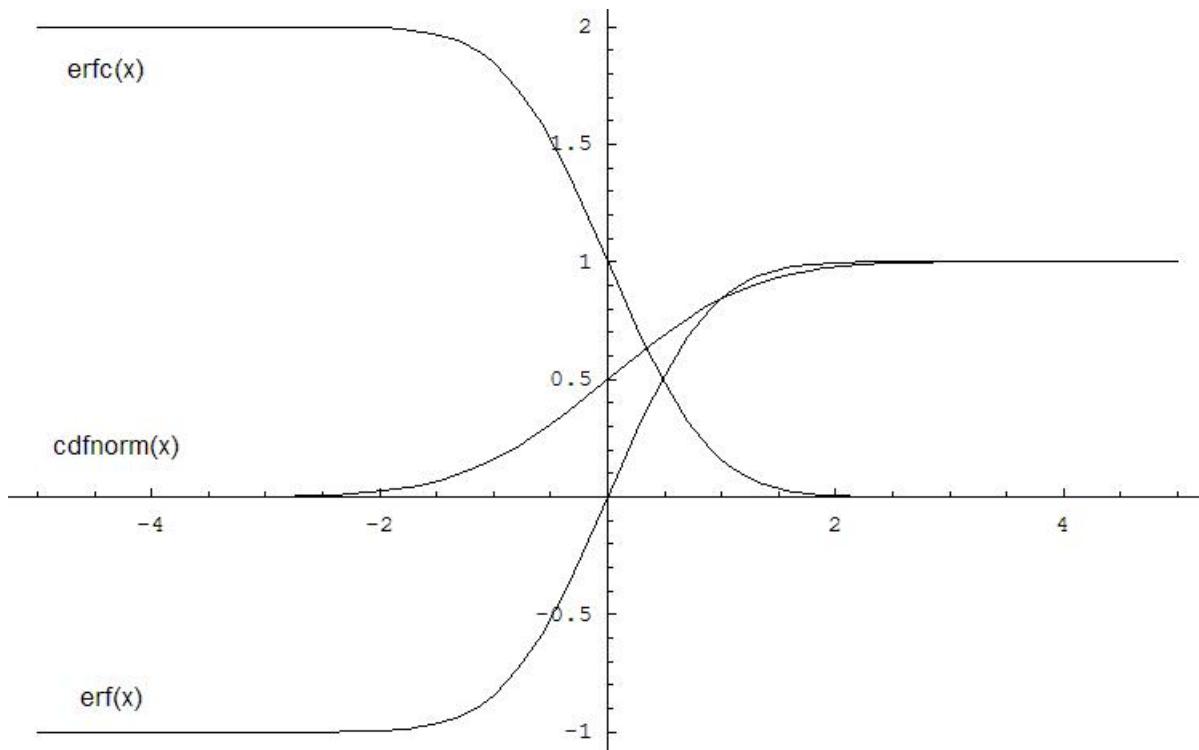
is the cumulative normal distribution function.

$$3. \Phi^{-1}(x) = \sqrt{2} \text{erf}^{-1}(2x - 1),$$

where $\Phi^{-1}(x)$ and $\text{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\text{erf}(x)$ respectively.

The following figure illustrates the relationships among `Erf`, `Erfc`, `CdfNorm`.

Erf Family Functions Relationship



Useful relations for these functions:

$$\text{erf}(x) + \text{erfc}(x) = 1$$

$$\text{cdfnorm}(x) = \frac{1}{2} \left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right) \right) = 1 - \frac{1}{2} \text{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

Special Values for Real Function v?Erf(x)

Argument	Result	Exception
+∞	+1	
-∞	-1	
QNAN	QNAN	
SNAN	QNAN	INVALID

See Also

[v?Erfc](#)
[v?CdfNorm](#)

v?Erfc

Computes the complementary error function value of vector elements.

Syntax**Fortran:**

```
call vserfc( n, a, y )
call vmserfc( n, a, y, mode )
call vderfc( n, a, y )
call vmderfc( n, a, y, mode )
```

C:

```
vsErfc( n, a, y );
vmsErfc( n, a, y, mode );
vdErfc( n, a, y );
vmdErfc( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
a	FORTRAN 77: REAL for vserfc, vmserfc DOUBLE PRECISION for vderfc, vmderfc Fortran 90: REAL, INTENT(IN) for vserfc, vmserfc DOUBLE PRECISION, INTENT(IN) for vderfc, vmderfc	Fortran: Array that specifies the input vector a. C: Pointer to an array that contains the input vector a.

Name	Type	Description
	C: const float* for vsErfc, vmsErfc const double* for vdErfc, vmdErfc	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vserfc, vmserfc DOUBLE PRECISION for vdserfc, vmdserfc Fortran 90: REAL, INTENT(OUT) for vserfc, vmserfc DOUBLE PRECISION, INTENT(OUT) for vdserfc, vmdserfc C: float* for vsErfc, vmsErfc double* for vdErfc, vmdErfc	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The `Erfc` function computes the complementary error function values for elements of the input vector `a` and writes them to the output vector `y`.

The complementary error function is defined as follows:

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_0^{\infty} e^{-t^2} dt$$

Useful relations:

1. $\text{erfc}(x) = 1 - \text{erf}(x)$.
2. $\Phi(x) = \frac{1}{2} \text{erf}\left(\frac{x}{\sqrt{2}}\right)$,

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

is the cumulative normal distribution function.

$$3. \Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1),$$

where $\Phi^{-1}(x)$ and $\operatorname{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\operatorname{erf}(x)$ respectively.

See also [Figure "Erf Family Functions Relationship"](#) in `Erf` function description for `Erfc` function relationship with the other functions of `Erf` family.

Special Values for Real Function v?Erfc(x)

Argument	Result	VML Error Status	Exception
X > underflow	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
$+\infty$	+0		
$-\infty$	+2		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[v?Erf](#)

[v?CdfNorm](#)

v?CdfNorm

Computes the cumulative normal distribution function values of vector elements.

Syntax

Fortran:

```
call vscdfnorm( n, a, y )
call vmscdfnorm( n, a, y, mode )
call vdcdfnorm( n, a, y )
call vmdcdfnorm( n, a, y, mode )
```

C:

```
vsCdfNorm( n, a, y );
vmsCdfNorm( n, a, y, mode );
vdCdfNorm( n, a, y );
vmdCdfNorm( n, a, y, mode );
```

Include Files

- Fortran: `mkl_vml.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>n</code>	<code>FORTTRAN 77: INTEGER</code> <code>Fortran 90: INTEGER, INTENT(IN)</code> <code>C: const MKL_INT</code>	Specifies the number of elements to be calculated.

Name	Type	Description
<i>a</i>	<p>FORTRAN 77: REAL for <code>vscdfnorm</code>, <code>vmscdfnorm</code></p> <p>DOUBLE PRECISION for <code>vdcdfnorm</code>, <code>vmdcdfnorm</code></p> <p>Fortran 90: REAL, INTENT(IN) for <code>vscdfnorm</code>, <code>vmscdfnorm</code></p> <p>DOUBLE PRECISION, INTENT(IN) for <code>vdcdfnorm</code>, <code>vmdcdfnorm</code></p> <p>C: const float* for <code>vsCdfNorm</code>, <code>vmsCdfNorm</code></p> <p>const double* for <code>vdCdfNorm</code>, <code>vmdCdfNorm</code></p>	<p>Fortran: Array that specifies the input vector <i>a</i>.</p> <p>C: Pointer to an array that contains the input vector <i>a</i>.</p>
<i>mode</i>	<p>FORTRAN 77: INTEGER*8</p> <p>Fortran 90: INTEGER(KIND=8), INTENT(IN)</p> <p>C: const MKL_INT64</p>	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<p>FORTRAN 77: REAL for <code>vscdfnorm</code>, <code>vmscdfnorm</code></p> <p>DOUBLE PRECISION for <code>vdcdfnorm</code>, <code>vmdcdfnorm</code></p> <p>Fortran 90: REAL, INTENT(OUT) for <code>vscdfnorm</code>, <code>vmscdfnorm</code></p> <p>DOUBLE PRECISION, INTENT(OUT) for <code>vdcdfnorm</code>, <code>vmdcdfnorm</code></p> <p>C: float* for <code>vsCdfNorm</code>, <code>vmsCdfNorm</code></p> <p>double* for <code>vdCdfNorm</code>, <code>vmdCdfNorm</code></p>	<p>Fortran: Array that specifies the output vector <i>y</i>.</p> <p>C: Pointer to an array that contains the output vector <i>y</i>.</p>

Description

The `CdfNorm` function computes the cumulative normal distribution function values for elements of the input vector *a* and writes them to the output vector *y*.

The cumulative normal distribution function is defined as given by:

$$\text{CdfNorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{t^2}{2}} dt .$$

Useful relations:

$$\text{cdfnorm}(x) = \frac{1}{2} \left(1 + \text{erf} \left(\frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \text{erfc} \left(\frac{x}{\sqrt{2}} \right)$$

where `Erf` and `Erfc` are the error and complementary error functions.

See also [Figure "Erf Family Functions Relationship"](#) in `Erf` function description for `CdfNorm` function relationship with the other functions of `Erf` family.

Special Values for Real Function v?CdfNorm(x)

Argument	Result	VML Error Status	Exception
X < underflow	+0	VML_STATUS_UNDERFLOW	UNDERFLOW
+∞	+1		
-∞	+0		
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[v?Erf](#)
[v?Erfc](#)

v?ErfInv

Computes inverse error function value of vector elements.

Syntax

Fortran:

```
call vserfinv( n, a, y )
call vmserfinv( n, a, y, mode )
call vderfinv( n, a, y )
call vmderfinv( n, a, y, mode )
```

C:

```
vsErfInv( n, a, y );
vmsErfInv( n, a, y, mode );
vdErfInv( n, a, y );
vmdErfInv( n, a, y, mode );
```

Include Files

- Fortran: `mkl_vml.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>n</code>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.

Name	Type	Description
	C: const MKL_INT	
a	FORTRAN 77: REAL for vserfinv, vmserfinv DOUBLE PRECISION for vderfinv, vmderfinv Fortran 90: REAL, INTENT(IN) for vserfinv, vmserfinv DOUBLE PRECISION, INTENT(IN) for vderfinv, vmderfinv C: const float* for vsErfInv, vmsErfInv const double* for vdErfInv, vmdErfInv	Fortran: Array that specifies the input vector a. C: Pointer to an array that contains the input vector a.
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vserfinv, vmserfinv DOUBLE PRECISION for vderfinv, vmderfinv Fortran 90: REAL, INTENT(OUT) for vserfinv, vmserfinv DOUBLE PRECISION, INTENT(OUT) for vderfinv, vmderfinv C: float* for vsErfInv, vmsErfInv double* for vdErfInv, vmdErfInv	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The `ErfInv` function computes the inverse error function values for elements of the input vector *a* and writes them to the output vector *y*

$$y = \text{erf}^{-1}(a),$$

where `erf(x)` is the error function defined as given by:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt .$$

Useful relations:

$$1. \quad \text{erf}^{-1}(x) = \text{erfc}^{-1}(1 - x),$$

where erfc is the complementary error function.

$$2. \quad \Phi(x) = \frac{1}{2} \text{erf}(x/\sqrt{2}),$$

where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp(-t^2/2) dt$$

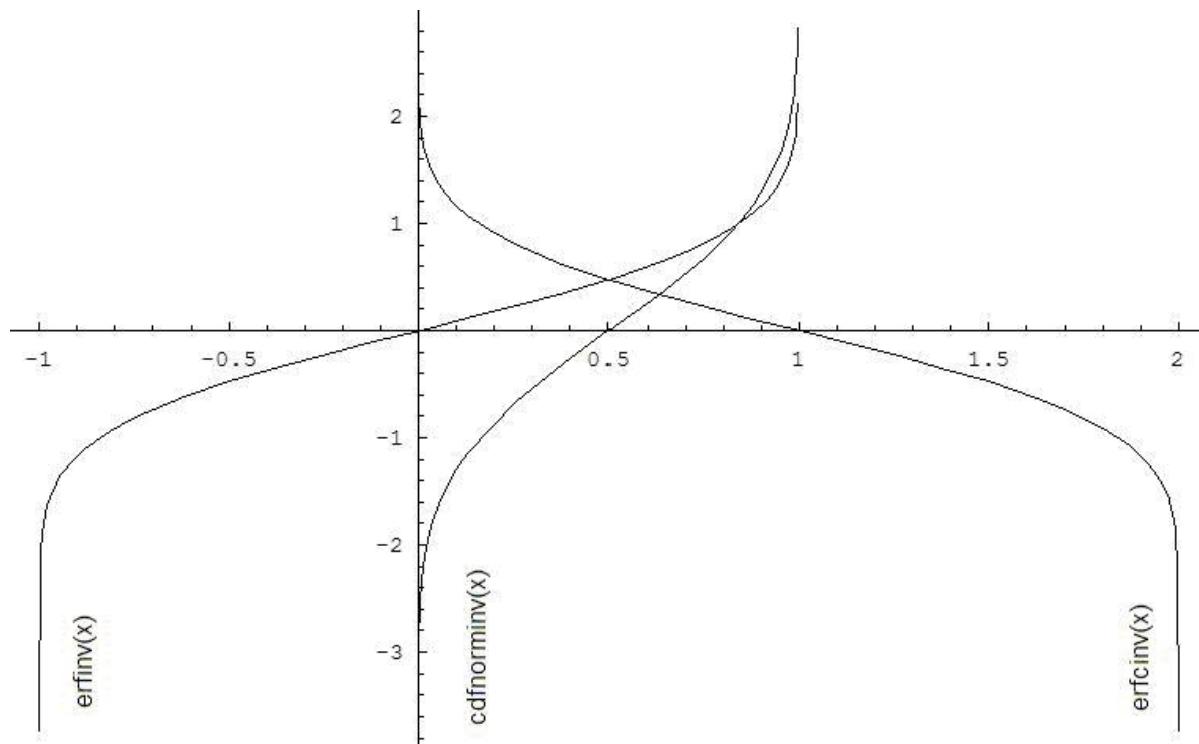
is the cumulative normal distribution function.

$$3. \quad \Phi^{-1}(x) = \sqrt{2} \text{erf}^{-1}(2x - 1),$$

where $\Phi^{-1}(x)$ and $\text{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $\text{erf}(x)$ respectively.

[Figure "ErfInv Family Functions Relationship"](#) illustrates the relationships among `ErfInv` family functions (`ErfInv`, `ErfcInv`, `CdfNormInv`).

[ErfInv Family Functions Relationship](#)



Useful relations for these functions:

erfcinv(x) = erfinv(1 - x)

$$\text{cdfnorminv}(x) = \sqrt{2}\text{erfinv}(2x - 1) = \sqrt{2}\text{erfcinv}(2 - 2x)$$

Special Values for Real Function v?ErfInv(x)

Argument	Result	VML Error Status	Exception
+0	+0		
-0	-0		
+1	+∞	VML_STATUS_SING	ZERODIVIDE
-1	-∞	VML_STATUS_SING	ZERODIVIDE
X > 1	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also[v?ErfcInv](#)[v?CdfNormInv](#)**v?ErfcInv**

Computes the inverse complementary error function value of vector elements.

Syntax**Fortran:**

```
call vserfcinv( n, a, y )
call vmserfcinv( n, a, y, mode )
call vderfcinv( n, a, y )
call vmderfcinv( n, a, y, mode )
```

C:

```
vsErfcInv( n, a, y );
vmsErfcInv( n, a, y, mode );
vdErfcInv( n, a, y );
vmdErfcInv( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN)	Specifies the number of elements to be calculated.

Name	Type	Description
	C: const MKL_INT	
<i>a</i>	FORTRAN 77: REAL for vserfcinv, vmserfcinv DOUBLE PRECISION for vderfcinv, vmderfcinv Fortran 90: REAL, INTENT(IN) for vserfcinv, vmserfcinv DOUBLE PRECISION, INTENT(IN) for vderfcinv, vmderfcinv C: const float* for vsErfcInv, vmsErfcInv const double* for vdErfcInv, vmdErfcInv	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vserfcinv, vmserfcinv DOUBLE PRECISION for vderfcinv, vmderfcinv Fortran 90: REAL, INTENT(OUT) for vserfcinv, vmserfcinv DOUBLE PRECISION, INTENT(OUT) for vderfcinv, vmderfcinv C: float* for vsErfcInv, vmsErfcInv double* for vdErfcInv, vmdErfcInv	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The `ErfcInv` function computes the inverse complimentary error function values for elements of the input vector *a* and writes them to the output vector *y*.

The inverse complementary error function is defined as given by:

$$\text{erfcinv}(x) = \text{erfinv}(1 - x)$$

$$\text{erfinv}(x) = \text{erf}^{-1}(x)$$

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

where $\text{erf}(x)$ denotes the error function and $\text{erfinv}(x)$ denotes the inverse error function.

See also [Figure "ErfInv Family Functions Relationship"](#) in [ErfInv](#) function description for [ErfcInv](#) function relationship with the other functions of [ErfInv](#) family.

Special Values for Real Function v?ErfcInv(x)

Argument	Result	VML Error Status	Exception
+1	+0		
+2	-∞	VML_STATUS_SING	ZERODIVIDE
-0	+∞	VML_STATUS_SING	ZERODIVIDE
+0	+∞	VML_STATUS_SING	ZERODIVIDE
X < -0	QNAN	VML_STATUS_ERRDOM	INVALID
X > +2	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[v?ErfInv](#)

[v?CdfNormInv](#)

v?CdfNormInv

Computes the inverse cumulative normal distribution function values of vector elements.

Syntax

Fortran:

```
call vscdfnorminv( n, a, y )
call vmscdfnorminv( n, a, y, mode )
call vdcdfnorminv( n, a, y )
call vmdcdfnorminv( n, a, y, mode )
```

C:

```
vsCdfNormInv( n, a, y );
vmsCdfNormInv( n, a, y, mode );
vdCdfNormInv( n, a, y );
vmdCdfNormInv( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER, INTENT(IN) <small>C:</small> const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	<small>FORTRAN 77:</small> REAL for vscdfnorminv, vmscdfnorminv <small>DOUBLE PRECISION for</small> vdcdfnorminv, vmdcdfnorminv <small>Fortran 90:</small> REAL, INTENT(IN) for vscdfnorminv, vmscdfnorminv <small>DOUBLE PRECISION, INTENT(IN) for</small> vdcdfnorminv, vmdcdfnorminv <small>C:</small> const float* for vsCdfNormInv, vmsCdfNormInv <small>const double* for</small> vdCdfNormInv, vmdCdfNormInv	<small>Fortran:</small> Array that specifies the input vector <i>a</i> . <small>C:</small> Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	<small>FORTRAN 77:</small> INTEGER*8 <small>Fortran 90:</small> INTEGER(KIND=8), INTENT(IN) <small>C:</small> const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<small>FORTRAN 77:</small> REAL for vscdfnorminv, vmscdfnorminv <small>DOUBLE PRECISION for</small> vdcdfnorminv, vmdcdfnorminv <small>Fortran 90:</small> REAL, INTENT(OUT) for vscdfnorminv, vmscdfnorminv <small>DOUBLE PRECISION, INTENT(OUT) for</small> vdcdfnorminv, vmdcdfnorminv <small>C:</small> float* for vsCdfNormInv, vmsCdfNormInv <small>double* for</small> vdCdfNormInv, vmdCdfNormInv	<small>Fortran:</small> Array that specifies the output vector <i>y</i> . <small>C:</small> Pointer to an array that contains the output vector <i>y</i> .

Description

The `CdfNormInv` function computes the inverse cumulative normal distribution function values for elements of the input vector a and writes them to the output vector y .

The inverse cumulative normal distribution function is defined as given by:

$$\text{CdfNormInv}(x) = \text{CdfNorm}^{-1}(x) ,$$

where `CdfNorm(x)` denotes the cumulative normal distribution function.

Useful relations:

$$\text{cdfnorminv}(x) = \sqrt{2}\text{erfinv}(2x - 1) = \sqrt{2}\text{erfcinv}(2 - 2x)$$

where `erfinv(x)` denotes the inverse error function and `erfcinv(x)` denotes the inverse complementary error functions.

See also [Figure "ErfInv Family Functions Relationship"](#) in `ErfInv` function description for `CdfNormInv` function relationship with the other functions of `ErfInv` family.

Special Values for Real Function v?CdfNormInv(x)

Argument	Result	VML Error Status	Exception
+0.5	+0		
+1	+∞	VML_STATUS_SING	ZERODIVIDE
-0	-∞	VML_STATUS_SING	ZERODIVIDE
+0	-∞	VML_STATUS_SING	ZERODIVIDE
X < -0	QNAN	VML_STATUS_ERRDOM	INVALID
X > +1	QNAN	VML_STATUS_ERRDOM	INVALID
+∞	QNAN	VML_STATUS_ERRDOM	INVALID
-∞	QNAN	VML_STATUS_ERRDOM	INVALID
QNAN	QNAN		
SNAN	QNAN		INVALID

See Also

[v?ErfInv](#)

[v?ErfcInv](#)

v?LGamma

Computes the natural logarithm of the absolute value of gamma function for vector elements.

Syntax

Fortran:

```
call vslgamma( n, a, y )
call vmslgamma( n, a, y, mode )
call vdlgamma( n, a, y )
call vmdlgamma( n, a, y, mode )
```

C:

```
vsLGamma( n, a, y );
vmsLGamma( n, a, y, mode );
vdLGamma( n, a, y );
vmdLGamma( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER, INTENT(IN) <small>C:</small> const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	<small>FORTRAN 77:</small> REAL for vslgamma, vmslgamma <small>DOUBLE PRECISION</small> for vdlgamma, vmdlgamma <small>Fortran 90:</small> REAL, INTENT(IN) for vs lgamma, vms lgamma <small>DOUBLE PRECISION, INTENT(IN)</small> for vd lgamma, vmd lgamma <small>C:</small> const float* for vsLGamma, vmsLGamma const double* for vdLGamma, vmdLGamma	<small>Fortran:</small> Array that specifies the input vector <i>a</i> . <small>C:</small> Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	<small>FORTRAN 77:</small> INTEGER*8 <small>Fortran 90:</small> INTEGER(KIND=8), INTENT(IN) <small>C:</small> const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<small>FORTRAN 77:</small> REAL for vslgamma, vmslgamma <small>DOUBLE PRECISION</small> for vdlgamma, vmdlgamma <small>Fortran 90:</small> REAL, INTENT(OUT) for vs lgamma, vms lgamma <small>DOUBLE PRECISION, INTENT(OUT)</small> for vd lgamma, vmd lgamma	<small>Fortran:</small> Array that specifies the output vector <i>y</i> . <small>C:</small> Pointer to an array that contains the output vector <i>y</i> .

Name	Type	Description
C:	float* for vsLGamma, vmsLGamma double* for vdLGamma, vmdLGamma	

Description

The `v?LGamma` function computes the natural logarithm of the absolute value of gamma function for elements of the input vector a and writes them to the output vector y . Precision overflow thresholds for the `v?LGamma` function are beyond the scope of this document. If the result does not meet the target precision, the function raises the `OVERFLOW` exception and sets the VML Error Status to `VML_STATUS_OVERFLOW`.

Special Values for Real Function `v?LGamma(x)`

Argument	Result	VML Error Status	Exception
+1	+0		
+2	+0		
+0	+?	VML_STATUS_SING	ZERODIVIDE
-0	+?	VML_STATUS_SING	ZERODIVIDE
negative integer	+?	VML_STATUS_SING	ZERODIVIDE
-?	+?		
+?	+?		
X > overflow	+?	VML_STATUS_OVERFLOW	OVERFLOW
QNAN	QNAN		
SNAN	QNAN		INVALID

`v?TGamma`

Computes the gamma function of vector elements.

Syntax

Fortran:

```
call vstgamma( n, a, y )
call vmstgamma( n, a, y, mode )
call vdtgamma( n, a, y )
call vmdtgamma( n, a, y, mode )
```

C:

```
vstGamma( n, a, y );
vmstGamma( n, a, y, mode );
vdtGamma( n, a, y );
vmdtGamma( n, a, y, mode );
```

Include Files

- Fortran: `mkl_vml.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77: REAL for vstgamma, vmstgamma DOUBLE PRECISION for vdtgamma, vmdtgamma Fortran 90: REAL, INTENT(IN) for vstgamma, vmstgamma DOUBLE PRECISION, INTENT(IN) for vdtgamma, vmdtgamma C: const float* for vsTGamma, vmsTGamma const double* for vdTGamma, vmdTGamma	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vstgamma, vmstgamma DOUBLE PRECISION for vdtgamma, vmdtgamma Fortran 90: REAL, INTENT(OUT) for vstgamma, vmstgamma DOUBLE PRECISION, INTENT(OUT) for vdtgamma, vmdtgamma C: float* for vsTGamma, vmsTGamma double* for vdTGamma, vmdTGamma	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?TGamma` function computes the gamma function for elements of the input vector *a* and writes them to the output vector *y*. Precision overflow thresholds for the `v?TGamma` function are beyond the scope of this document. If the result does not meet the target precision, the function raises the `OVERFLOW` exception and sets the VML Error Status to `VML_STATUS_OVERFLOW`.

Special Values for Real Function v?TGamma(x)

Argument	Result	VML Error Status	Exception
+0	$+\infty$	VML_STATUS_SING	ZERODIVIDE
-0	$-\infty$	VML_STATUS_SING	ZERODIVIDE
negative integer	QNAN	VML_STATUS_ERRDOM	INVALID
$-\infty$	QNAN	VML_STATUS_ERRDOM	INVALID
$+\infty$	$+\infty$		
$X > \text{overflow}$	$+\infty$	VML_STATUS_OVERFLOW	OVERFLOW
QNAN	QNAN		
SNAN	QNAN		INVALID

Rounding Functions

v?Floor

Computes an integer value rounded towards minus infinity for each vector element.

Syntax

Fortran:

```
call vsfloor( n, a, y )
call vmsfloor( n, a, y, mode )
call vdfloor( n, a, y )
call vmdfloor( n, a, y, mode )
```

C:

```
vsFloor( n, a, y );
vmsFloor( n, a, y, mode );
vdFloor( n, a, y );
vmdFloor( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
a	FORTTRAN 77: REAL for vsfloor, vmsfloor DOUBLE PRECISION for vdfloor, vmdfloor	Fortran: Array that specifies the input vector a. C: Pointer to an array that contains the input vector a.

Name	Type	Description
	Fortran 90: REAL, INTENT(IN) for vsfloor, vmsfloor DOUBLE PRECISION, INTENT(IN) for vdfloor, vmdfloor C: const float* for vsFloor, vmsFloor const double* for vdFloor, vmdFloor	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vsfloor, vmsfloor DOUBLE PRECISION for vdfloor, vmdfloor Fortran 90: REAL, INTENT(OUT) for vsfloor, vmsfloor DOUBLE PRECISION, INTENT(OUT) for vdfloor, vmdfloor C: float* for vsFloor, vmsFloor double* for vdFloor, vmdFloor	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The function computes an integer value rounded towards minus infinity for each vector element.

$$y_i = \lfloor a_i \rfloor$$

Special Values for Real Function v?Floor(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Ceil

Computes an integer value rounded towards plus infinity for each vector element.

Syntax

Fortran:

```
call vsceil( n, a, y )
call vmsceil( n, a, y, mode )
call vdceil( n, a, y )
call vmdceil( n, a, y, mode )
```

C:

```
vsCeil( n, a, y );
vmsCeil( n, a, y, mode );
vdCeil( n, a, y );
vmdCeil( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTAN 77: REAL for vsceil, vmsceil DOUBLE PRECISION for vdceil, vmdceil Fortran 90: REAL, INTENT(IN) for vsceil, vmsceil DOUBLE PRECISION, INTENT(IN) for vdceil, vmdceil C: const float* for vsCeil, vmsCeil const double* for vdCeil, vmdCeil	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vsceil, vmsceil DOUBLE PRECISION for vdceil, vmdceil Fortran 90: REAL, INTENT(OUT) for vsceil, vmsceil DOUBLE PRECISION, INTENT(OUT) for vdceil, vmdceil C: float* for vsCeil, vmsCeil double* for vdCeil, vmdCeil	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The function computes an integer value rounded towards plus infinity for each vector element.

$$y_i = \lceil a_i \rceil$$

Special Values for Real Function v?Ceil(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Trunc

Computes an integer value rounded towards zero for each vector element.

Syntax

Fortran:

```
call vstrunc( n, a, y )
call vmstrunc( n, a, y, mode )
call vdtrunc( n, a, y )
call vmdtrunc( n, a, y, mode )
```

C:

```
vsTrunc( n, a, y );
vmsTrunc( n, a, y, mode );
vdTrunc( n, a, y );
vmdTrunc( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER, INTENT(IN) <small>C:</small> const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	<small>FORTRAN 77:</small> REAL for vsTrunc, <small>vmsTrunc</small> <small>DOUBLE PRECISION</small> for vdTrunc, <small>vmdTrunc</small> <small>Fortran 90:</small> REAL, INTENT(IN) for <small>vsTrunc, vmsTrunc</small> <small>DOUBLE PRECISION, INTENT(IN)</small> for <small>vdTrunc, vmdTrunc</small> <small>C:</small> const float* for vsTrunc, <small>vmsTrunc</small> <small>const double*</small> for vdTrunc, <small>vmdTrunc</small>	<small>Fortran:</small> Array that specifies the input vector <i>a</i> . <small>C:</small> Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	<small>FORTRAN 77:</small> INTEGER*8 <small>Fortran 90:</small> INTEGER(KIND=8), <small>INTENT(IN)</small> <small>C:</small> const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	<small>FORTRAN 77:</small> REAL for vsTrunc, <small>vmsTrunc</small> <small>DOUBLE PRECISION</small> for vdTrunc, <small>vmdTrunc</small> <small>Fortran 90:</small> REAL, INTENT(OUT) for <small>vsTrunc, vmsTrunc</small> <small>DOUBLE PRECISION, INTENT(OUT)</small> for <small>vdTrunc, vmdTrunc</small> <small>C:</small> float* for vsTrunc, vmsTrunc <small>double*</small> for vdTrunc, vmdTrunc	<small>Fortran:</small> Array that specifies the output vector <i>y</i> . <small>C:</small> Pointer to an array that contains the output vector <i>y</i> .

Description

The function computes an integer value rounded towards zero for each vector element.

$$a_i \geq 0, y_i = \lfloor a_i \rfloor$$

$$a_i < 0, y_i = \lceil a_i \rceil$$

Special Values for Real Function v?Trunc(x)

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Round

Computes a value rounded to the nearest integer for each vector element.

Syntax

Fortran:

```
call vsround( n, a, y )
call vmsround( n, a, y, mode )
call vdround( n, a, y )
call vmdround( n, a, y, mode )
```

C:

```
vsRound( n, a, y );
vmsRound( n, a, y, mode );
vdRound( n, a, y );
vmdRound( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	FORTTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
a	FORTTRAN 77: REAL for vsround, vmsround DOUBLE PRECISION for vdround, vmdround	Fortran: Array that specifies the input vector a. C: Pointer to an array that contains the input vector a.

Name	Type	Description
	<p>Fortran 90: REAL, INTENT(IN) for vsround, vmsround</p> <p>DOUBLE PRECISION, INTENT(IN) for vdround, vmdround</p> <p>C: const float* for vsRound, vmsRound</p> <p>const double* for vdRound, vmdRound</p>	
mode	<p>FORTRAN 77: INTEGER*8</p> <p>Fortran 90: INTEGER(KIND=8), INTENT(IN)</p> <p>C: const MKL_INT64</p>	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	<p>FORTRAN 77: REAL for vsround, vmsround</p> <p>DOUBLE PRECISION for vdround, vmdround</p> <p>Fortran 90: REAL, INTENT(OUT) for vsround, vmsround</p> <p>DOUBLE PRECISION, INTENT(OUT) for vdround, vmdround</p> <p>C: float* for vsRound, vmsRound double* for vdRound, vmdRound</p>	<p>Fortran: Array that specifies the output vector y.</p> <p>C: Pointer to an array that contains the output vector y.</p>

Description

The function computes a value rounded to the nearest integer for each vector element. Input elements that are halfway between two consecutive integers are always rounded away from zero regardless of the rounding mode.

Special Values for Real Function v?Round(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?NearbyInt

Computes a rounded integer value in the current rounding mode for each vector element.

Syntax

Fortran:

```
call vsnearbyint( n, a, y )
call vmsnearbyint( n, a, y, mode )
call vdnearbyint( n, a, y )
call vmdnearbyint( n, a, y, mode )
```

C:

```
vsNearbyInt( n, a, y );
vmsNearbyInt( n, a, y, mode );
vdNearbyInt( n, a, y );
vmdNearbyInt( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTTRAN 77: REAL for vsnearbyint, vmsnearbyint DOUBLE PRECISION for vdnearbyint, vmdnearbyint Fortran 90: REAL, INTENT(IN) for vsnearbyint, vmsnearbyint DOUBLE PRECISION, INTENT(IN) for vdnearbyint, vmdnearbyint C: const float* for vsNearbyInt, vmsNearbyInt const double* for vdNearbyInt, vmdNearbyInt	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vsnearbyint, vmsnearbyint DOUBLE PRECISION for vdnearbyint, vmdnearbyint	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .
	Fortran 90: REAL, INTENT(OUT) for vsnearbyint, vmsnearbyint	
	DOUBLE PRECISION, INTENT(OUT) for vdnearbyint, vmdnearbyint	
	C: float* for vsNearbyInt, vmsNearbyInt	
	double* for vdNearbyInt, vmdNearbyInt	

Description

The *v?NearbyInt* function computes a rounded integer value in a current rounding mode for each vector element.

Special Values for Real Function *v?NearbyInt(x)*

Argument	Argument	Exception
+0	+0	
-0	-0	
+∞	+∞	
-∞	-∞	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Rint

Computes a rounded integer value in the current rounding mode.

Syntax

Fortran:

```
call vsrint( n, a, y )
call vmsrint( n, a, y, mode )
call vdrint( n, a, y )
call vmdrint( n, a, y, mode )
```

C:

```
vsRint( n, a, y );
vmsRint( n, a, y, mode );
vdRint( n, a, y );
vmdRint( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77: REAL for vsrint, vmsrint DOUBLE PRECISION for vdrint, vmdrint Fortran 90: REAL, INTENT(IN) for vsrint, vmsrint DOUBLE PRECISION, INTENT(IN) for vdrint, vmdrint C: const float* for vsRint, vmsRint const double* for vdRint, vmdRint	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .
<i>mode</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
<i>y</i>	FORTRAN 77: REAL for vsrint, vmsrint DOUBLE PRECISION for vdrint, vmdrint Fortran 90: REAL, INTENT(OUT) for vsrint, vmsrint DOUBLE PRECISION, INTENT(OUT) for vdrint, vmdrint C: float* for vsRint, vmsRint double* for vdRint, vmdRint	Fortran: Array that specifies the output vector <i>y</i> . C: Pointer to an array that contains the output vector <i>y</i> .

Description

The `v?Rint` function computes a rounded floating-point integer value using the current rounding mode for each vector element.

The rounding mode affects the results computed for inputs that fall between consecutive integers. For example:

- $f(0.5) = 0$, for rounding modes set to round to nearest round toward zero or to minus infinity.
- $f(0.5) = 1$, for rounding modes set to plus infinity.
- $f(-1.5) = -2$, for rounding modes set to round to nearest or to minus infinity.
- $f(-1.5) = -1$, for rounding modes set to round toward zero or to plus infinity.

Special Values for Real Function v?Rint(x)

Argument	Result	Exception
+0	+0	
-0	-0	
$+\infty$	$+\infty$	
$-\infty$	$-\infty$	
SNAN	QNAN	INVALID
QNAN	QNAN	

v?Modf

Computes a truncated integer value and the remaining fraction part for each vector element.

Syntax

Fortran:

```
call vsmodf( n, a, y, z )
call vmsmodf( n, a, y, z, mode )
call vdmodf( n, a, y, z )
call vmdmodf( n, a, y, z, mode )
```

C:

```
vsModf( n, a, y, z );
vmsModf( n, a, y, z, mode );
vdModf( n, a, y, z );
vmdModf( n, a, y, z, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77: REAL for vsmodf, vmsmodf	Fortran: Array, specifies the input vector <i>a</i> .

Name	Type	Description
	DOUBLE PRECISION for vdmodf, vmdmodf	C: Pointer to an array that contains the input vector a .
	Fortran 90: REAL, INTENT(IN) for vsmodf, vmsmodf	
	DOUBLE PRECISION, INTENT(IN) for vdmodf, vmdmodf	
	C: const float* for vsModf, vmsModf	
	const double* for vdModf, vmdModf	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y, z	FORTRAN 77: REAL for vsmodf, vmsmodf	Fortran: Array, specifies the output vector y and z .
	DOUBLE PRECISION for vdmodf, vmdmodf	C: Pointer to an array that contains the output vector y and z .
	Fortran 90: REAL, INTENT(OUT) for vsmodf, vmsmodf	
	DOUBLE PRECISION, INTENT(OUT) for vdmodf, vmdmodf	
	C: float* for vsModf, vmsModf	
	double* for vdModf, vmdModf	

Description

The function computes a truncated integer value and the remaining fraction part for each vector element.

$$a_i \geq 0, \begin{cases} y_i = \lfloor a_i \rfloor \\ z_i = a_i - \lfloor a_i \rfloor \end{cases}$$

$$a_i < 0, \begin{cases} y_i = \lceil a_i \rceil \\ z_i = a_i - \lceil a_i \rceil \end{cases}$$

Special Values for Real Function v?Modf(x)

Argument	Result: $y(i)$	Result: $z(i)$	Exception
+0	+0	+0	

Argument	Result: $y(i)$	Result: $z(i)$	Exception
-0	-0	-0	
$+\infty$	$+\infty$	$+0$	
$-\infty$	$-\infty$	-0	
SNAN	QNAN	QNAN	INVALID
QNAN	QNAN	QNAN	

v?Frac

Computes a signed fractional part for each vector element.

Syntax

Fortran:

```
call vsfrac( n, a, y )
call vmsfrac( n, a, y, mode )
call vdfrac( n, a, y )
call vmdfrac( n, a, y, mode )
```

C:

```
vsFrac( n, a, y );
vmsFrac( n, a, y, mode );
vdFrac( n, a, y );
vmdFrac( n, a, y, mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the number of elements to be calculated.
<i>a</i>	FORTRAN 77: REAL for vsfrac, vmsfrac DOUBLE PRECISION for vdfrac, vmdfrac Fortran 90: REAL, INTENT(IN) for vsfrac, vmsfrac DOUBLE PRECISION, INTENT(IN) for vdfrac, vmdfrac C: const float* for vsFrac, vmsFrac	Fortran: Array that specifies the input vector <i>a</i> . C: Pointer to an array that contains the input vector <i>a</i> .

Name	Type	Description
	const double* for vdFrac, vmdFrac	
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_INT64	Overrides global VML mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name	Type	Description
y	FORTRAN 77: REAL for vsfrac, vmsfrac DOUBLE PRECISION for vdfrac, vmdfrac Fortran 90: REAL, INTENT(OUT) for vsfrac, vmsfrac DOUBLE PRECISION, INTENT(OUT) for vdfrac, vmdfrac C: float* for vsFrac, vmsFrac double* for vdFrac, vmdFrac	Fortran: Array that specifies the output vector y. C: Pointer to an array that contains the output vector y.

Description

The function computes a signed fractional part for each vector element.

$$y_i = \begin{cases} a_i - \lfloor a_i \rfloor, & a_i \geq 0 \\ a_i - \lceil a_i \rceil, & a_i < 0 \end{cases}$$

Special Values for Real Function v?Frac(x)

Argument	Result	Exception
+0	+0	
-0	-0	
+∞	+0	
-∞	-0	
SNAN	QNAN	INVALID
QNAN	QNAN	

VML Pack/Unpack Functions

This section describes VML functions that convert vectors with unit increment to and from vectors with positive increment indexing, vector indexing, and mask indexing (see Appendix B for details on vector indexing methods).

The table below lists available VML Pack/Unpack functions, together with data types and indexing methods associated with them.

VML Pack/Unpack Functions

Function Short Name	Data Types	Indexing Methods	Description
v?Pack	s, d, c, z	I,V,M	Gathers elements of arrays, indexed by different methods.
v?Unpack	s, d, c, z	I,V,M	Scatters vector elements to arrays with different indexing.

See Also

[Vector Arguments in VML](#)

v?Pack

Copies elements of an array with specified indexing to a vector with unit increment.

Syntax**Fortran:**

```
call vspacki( n, a, inca, y )
call vspackv( n, a, ia, y )
call vspackm( n, a, ma, y )
call vdpacki( n, a, inca, y )
call vdpackv( n, a, ia, y )
call vdpackm( n, a, ma, y )
call vcpacki( n, a, inca, y )
call vcpackv( n, a, ia, y )
call vcpackm( n, a, ma, y )
call vzpacki( n, a, inca, y )
call vzpackv( n, a, ia, y )
call vzpackm( n, a, ma, y )
```

C:

```
vsPackI( n, a, inca, y );
vsPackV( n, a, ia, y );
vsPackM( n, a, ma, y );
vdPackI( n, a, inca, y );
vdPackV( n, a, ia, y );
vdPackM( n, a, ma, y );
vcPackI( n, a, inca, y );
vcPackV( n, a, ia, y );
vcPackM( n, a, ma, y );
vzPackI( n, a, inca, y );
vzPackV( n, a, ia, y );
vzPackM( n, a, ma, y );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER, INTENT(IN)</p> <p>C: const MKL_INT</p>	Specifies the number of elements to be calculated.
<i>a</i>	<p>FORTRAN 77: REAL for vspacki, vspackv, vspackm</p> <p>DOUBLE PRECISION for vdpacki, vdpackv, vdpackm</p> <p>COMPLEX for vcpacki, vcpackv, vcpackm</p> <p>DOUBLE COMPLEX for vzpacki, vzpackv, vzpackm</p> <p>Fortran 90: REAL, INTENT(IN) for vspacki, vspackv, vspackm</p> <p>DOUBLE PRECISION, INTENT(IN) for vdpacki, vdpackv, vdpackm</p> <p>COMPLEX, INTENT(IN) for vcpacki, vcpackv, vcpackm</p> <p>DOUBLE COMPLEX, INTENT(IN) for vzpacki, vzpackv, vzpackm</p> <p>C: const float* for vsPackI, vsPackV, vsPackM</p> <p>const double* for vdPackI, vdPackV, vdPackM</p> <p>C: const MKL_Complex8* for vcPackI, vcPackV, vcPackM</p> <p>const MKL_Complex16* for vzPackI, vzPackV, vzPackM</p>	<p>Fortran: Array, DIMENSION at least(1 + (n-1)*inca) for v?packi, Array, DIMENSION at least max(n,max(ia[j])), j=0, ..., n-1 for v?packv, Array, DIMENSION at least n for v?packm.</p> <p>Specifies the input vector <i>a</i>.</p> <p>C: Specifies pointer to an array that contains the input vector <i>a</i>. The arrays must be:</p> <p>for v?PackI, at least(1 + (n-1)*inca)</p> <p>for v?PackV, at least max(n,max(ia[j])), j=0, ..., n-1</p> <p>for v?PackM, at least n.</p>
<i>inca</i>	<p>FORTRAN 77: INTEGER for vspacki, vdpacki, vcpacki, vzpacki</p> <p>Fortran 90: INTEGER, INTENT(IN) for vspacki, vdpacki, vcpacki, vzpacki</p> <p>C: const MKL_INT for vsPackI, vdPackI, vcPackI, vzPackI</p>	Specifies the increment for the elements of <i>a</i> .

Name	Type	Description
<i>ia</i>	<p>FORTRAN 77: INTEGER for vspackv, vdpackv, vcpackv, vzpackv</p> <p>Fortran 90: INTEGER, INTENT(IN) for vspackv, vdpackv, vcpackv, vzpackv</p> <p>C: const int* for vsPackV, vdPackV, vcPackV, vzPackV</p>	<p>Fortran: Array, DIMENSION at least <i>n</i>. Specifies the index vector for the elements of <i>a</i>.</p> <p>C: Specifies the pointer to an array of size at least <i>n</i> that contains the index vector for the elements of <i>a</i>.</p>
<i>ma</i>	<p>FORTRAN 77: INTEGER for vspackm, vdpackm, vcpackm, vzpackm</p> <p>Fortran 90: INTEGER, INTENT(IN) for vspackm, vdpackm, vcpackm, vzpackm</p> <p>C: const int* for vsPackM, vdPackM, vcPackM, vzPackM</p>	<p>Fortran: Array, DIMENSION at least <i>n</i>, Specifies the mask vector for the elements of <i>a</i>.</p> <p>C: Specifies the pointer to an array of size at least <i>n</i> that contains the mask vector for the elements of <i>a</i>.</p>

Output Parameters

Name	Type	Description
<i>y</i>	<p>FORTRAN 77: REAL for vspacki, vspackv, vspackm</p> <p>DOUBLE PRECISION for vdpacki, vdpackv, vdpackm</p> <p>COMPLEX for vcpacki, vcpackv, vcpackm</p> <p>DOUBLE COMPLEX for vzpacki, vzpackv, vzpackm</p> <p>Fortran 90: REAL, INTENT(OUT) for vspacki, vspackv, vspackm</p> <p>DOUBLE PRECISION, INTENT(OUT) for vdpacki, vdpackv, vdpackm</p> <p>COMPLEX, INTENT(OUT) for vcpacki, vcpackv, vcpackm</p> <p>DOUBLE COMPLEX, INTENT(OUT) for vzpacki, vzpackv, vzpackm</p> <p>C: float* for vsPackI, vsPackV, vsPackM</p> <p>double* for vdPackI, vdPackV, vdPackM</p> <p>const MKL_Complex8* for vcPackI, vcPackV, vcPackM</p>	<p>Fortran: Array, DIMENSION at least <i>n</i>. Specifies the output vector <i>y</i>.</p> <p>C: Pointer to an array of size at least <i>n</i> that contains the output vector <i>y</i>.</p>

Name	Type	Description
	const MKL_Complex16* for vzPackI, vzPackV, vzPackM	

v?Unpack

Copies elements of a vector with unit increment to an array with specified indexing.

Syntax

Fortran:

```
call vsunpacki( n, a, y, incy )
call vsunpackv( n, a, y, iy )
call vsunpackm( n, a, y, my )
call vdunpacki( n, a, y, incy )
call vdunpackv( n, a, y, iy )
call vdunpackm( n, a, y, my )
call vcunpacki( n, a, y, incy )
call vcunpackv( n, a, y, iy )
call vcunpackm( n, a, y, my )
call vzunpacki( n, a, y, incy )
call vzunpackv( n, a, y, iy )
call vzunpackm( n, a, y, my )
```

C:

```
vsUnpackI( n, a, y, incy );
vsUnpackV( n, a, y, iy );
vsUnpackM( n, a, y, my );
vdUnpackI( n, a, y, incy );
vdUnpackV( n, a, y, iy );
vdUnpackM( n, a, y, my );
vcUnpackI( n, a, y, incy );
vcUnpackV( n, a, y, iy );
vcUnpackM( n, a, y, my );
vzUnpackI( n, a, y, incy );
vzUnpackV( n, a, y, iy );
vzUnpackM( n, a, y, my );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER, INTENT(IN)</p> <p>C: const MKL_INT</p>	Specifies the number of elements to be calculated.
<i>a</i>	<p>FORTRAN 77: REAL for vsunpacki, vsunpackv, vsunpackm</p> <p>DOUBLE PRECISION for vdunpacki, vdunpackv, vdunpackm</p> <p>COMPLEX for vcunpacki, vcunpackv, vcunpackm</p> <p>DOUBLE COMPLEX for vzunpacki, vzunpackv, vzunpackm</p> <p>Fortran 90: REAL, INTENT(IN) for vsunpacki, vsunpackv, vsunpackm</p> <p>DOUBLE PRECISION, INTENT(IN) for vdunpacki, vdunpackv, vdunpackm</p> <p>COMPLEX, INTENT(IN) for vcunpacki, vcunpackv, vcunpackm</p> <p>DOUBLE COMPLEX, INTENT(IN) for vzunpacki, vzunpackv, vzunpackm</p> <p>C: const float* for vsUnpackI, vsUnpackV, vsUnpackM</p> <p>const double* for vdUnpackI, vdUnpackV, vdUnpackM</p> <p>const MKL_Complex8* for vcUnpackI, vcUnpackV, vcUnpackM</p> <p>const MKL_Complex16* for vzUnpackI, vzUnpackV, vzUnpackM</p>	<p>Fortran: Array, DIMENSION at least <i>n</i>.</p> <p>Specifies the input vector <i>a</i>.</p> <p>C: Specifies the pointer to an array of size at least <i>n</i> that contains the input vector <i>a</i>.</p>
<i>incy</i>	<p>FORTRAN 77: INTEGER for vsunpacki, vdunpacki, vcunpacki, vzunpacki</p> <p>Fortran 90: INTEGER, INTENT(IN) for vsunpacki, vdunpacki, vcunpacki, vzunpacki</p> <p>C: const MKL_INT for vsUnpackI, vdUnpackI, vcUnpackI, vzUnpackI</p>	Specifies the increment for the elements of <i>y</i> .
<i>iy</i>	<p>FORTRAN 77: INTEGER for vsunpackv, vdunpackv, vcunpackv, vzunpackv</p>	<p>Fortran: Array, DIMENSION at least <i>n</i>.</p> <p>Specifies the index vector for the elements of <i>y</i>.</p> <p>C: Specifies the pointer to an array of size at least <i>n</i> that contains the index vector for the elements of <i>a</i>.</p>

Name	Type	Description
	<pre> Fortran 90: INTEGER, INTENT(IN) for vsunpackv, vdunpackv, vcunpackv, vzunpackv C: const int* for vsUnpackV, vdUnpackV, vcUnpackV, vzUnpackV </pre>	
my	<pre> FORTRAN 77: INTEGER for vsunpackm, vdunpackm, vcunpackm, vzunpackm Fortran 90: INTEGER, INTENT(IN) for vsunpackm, vdunpackm, vcunpackm, vzunpackm C: const int* for vsUnpackM, vdUnpackM, vcUnpackM, vzUnpackM </pre>	<p>Fortran: Array, DIMENSION at least n, Specifies the mask vector for the elements of y.</p> <p>C: Specifies the pointer to an array of size at least n that contains the mask vector for the elements of a.</p>

Output Parameters

Name	Type	Description
y	<pre> FORTRAN 77: REAL for vsunpacki, vsunpackv, vsunpackm DOUBLE PRECISION for vdunpacki, vdunpackv, vdunpackm COMPLEX, INTENT(IN) for vcunpacki, vcunpackv, vcunpackm DOUBLE COMPLEX, INTENT(IN) for vzunpacki, vzunpackv, vzunpackm Fortran 90: REAL, INTENT(OUT) for vsunpacki, vsunpackv, vsunpackm DOUBLE PRECISION, INTENT(OUT) for vdunpacki, vdunpackv, vdunpackm COMPLEX, INTENT(OUT) for vcunpacki, vcunpackv, vcunpackm DOUBLE COMPLEX, INTENT(OUT) for vzunpacki, vzunpackv, vzunpackm C: float* for vsUnpackI, vsUnpackV, vsUnpackM double* for vdUnpackI, vdUnpackV, vdUnpackM const MKL_Complex8* for vcUnpackI, vcUnpackV, vcUnpackM const MKL_Complex16* for vzUnpackI, vzUnpackV, vzUnpackM </pre>	<p>Fortran: Array, DIMENSION for v?unpacki, at least $(1 + (n-1)*incy)$ for v?unpackv, at least $\max(n, \max(iy[j]))$, $j=0, \dots, n-1$ for v?unpackm, at least n</p> <p>C: Specifies the pointer to an array that contains the output vector y. The array must be: for v?UnpackI, at least $(1 + (n-1)*incy)$ for v?UnpackV, at least $\max(n, \max(ia[j]))$, $j=0, \dots, n-1$, for v?UnpackM, at least n.</p>

VML Service Functions

The VML Service functions enable you to set/get the accuracy mode and error code. These functions are available both in the Fortran and C interfaces. The table below lists available VML Service functions and their short description.

VML Service Functions

Function Short Name	Description
vmlSetMode	Sets the VML mode
vmlGetMode	Gets the VML mode
MKLFreeTls	Frees allocated VML/VSL thread local storage memory from within DllMain routine (Windows* OS only)
vmlSetErrStatus	Sets the VML Error Status
vmlGetErrStatus	Gets the VML Error Status
vmlClearErrStatus	Clears the VML Error Status
vmlSetErrorHandler	Sets the additional error handler callback function
vmlGetErrorHandler	Gets the additional error handler callback function
vmlClearErrorHandler	Deletes the additional error handler callback function

vmlSetMode

Sets a new mode for VML functions according to the mode parameter and stores the previous VML mode to oldmode.

Syntax

Fortran:

```
oldmode = vmlsetmode( mode )
```

C:

```
oldmode = vmlSetMode( mode );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
mode	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER(KIND=8), INTENT(IN) C: const MKL_UINT	Specifies the VML mode to be set.

Output Parameters

Name	Type	Description
oldmode	Fortran: INTEGER*8 Fortran 90: INTEGER(KIND=8) C: unsigned int	Specifies the former VML mode.

Description

The `vmlSetMode` function sets a new mode for VML functions according to the *mode* parameter and stores the previous VML mode to *oldmode*. The mode change has a global effect on all the VML functions within a thread.

NOTE

You can override the global mode setting and change the mode for a given VML function call by using a respective `vm[s,d]<Func>` variant of the function.

The *mode* parameter is designed to control accuracy, handling of denormalized numbers, and error handling. Table "Values of the *mode* Parameter" lists values of the *mode* parameter. You can obtain all other possible values of the *mode* parameter values by using bitwise OR (|) operation to combine one value for accuracy, one value for handling of denormalized numbers, and one value for error control options. The default value of the *mode* parameter is `VML_HA | VML_FTZDAZ_OFF | VML_ERRMODE_DEFAULT`.

The `VML_FTZDAZ_ON` mode is specifically designed to improve the performance of computations that involve denormalized numbers at the cost of reasonable accuracy loss. This mode changes the numeric behavior of the functions: denormalized input values are treated as zeros (`DAZ` = denormals-are-zero) and denormalized results are flushed to zero (`FTZ` = flush-to-zero). Accuracy loss may occur if input and/or output values are close to denormal range.

Values of the *mode* Parameter

Value of <i>mode</i>	Description
Accuracy Control	
<code>VML_HA</code>	high accuracy versions of VML functions
<code>VML_LA</code>	low accuracy versions of VML functions
<code>VML_EP</code>	enhanced performance accuracy versions of VML functions
Denormalized Numbers Handling Control	
<code>VML_FTZDAZ_ON</code>	Faster processing of denormalized inputs is enabled.
<code>VML_FTZDAZ_OFF</code>	Faster processing of denormalized inputs is disabled.
Error Mode Control	
<code>VML_ERRMODE_IGNORE</code>	No action is set for computation errors.
<code>VML_ERRMODE_ERRNO</code>	On error, the <code>errno</code> variable is set.
<code>VML_ERRMODE_STDERR</code>	On error, the error text information is written to <code>stderr</code> .
<code>VML_ERRMODE_EXCEPT</code>	On error, an exception is raised.
<code>VML_ERRMODE_CALLBACK</code>	On error, an additional error handler function is called.
<code>VML_ERRMODE_DEFAULT</code>	On error, the <code>errno</code> variable is set, an exception is raised, and an additional error handler function is called.

Examples

The following example shows how to set low accuracy, fast processing for denormalized numbers and `stderr` error mode:

```
oldmode = vmlsetmode( VML_LA )
call vmlsetmode( IOR(VML_LA, VML_FTZDAZ_ON, VML_ERRMODE_STDERR) )

vmlSetMode( VML_LA );
vmlSetMode( VML_LA | VML_FTZDAZ_ON | VML_ERRMODE_STDERR );
```

vmlGetMode

Gets the VML mode.

Syntax

Fortran:

```
mod = vmlgetmode()
```

C:

```
mod = vmlGetMode( void );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Output Parameters

Name	Type	Description
mod	Fortran: INTEGER C: unsigned int	Specifies the packed <i>mode</i> parameter.

Description

The function `vmlGetMode()` returns the VML *mode* parameter that controls accuracy, handling of denormalized numbers, and error handling options. The *mod* variable value is a combination of the values listed in the table "[Values of the *mode* Parameter](#)". You can obtain these values using the respective mask from the table "[Values of Mask for the *mode* Parameter](#)".

Values of Mask for the *mode* Parameter

Value of mask	Description
VML_ACCURACY_MASK	Specifies mask for accuracy <i>mode</i> selection.
VML_FTZDAZ_MASK	Specifies mask for FTZDAZ <i>mode</i> selection.
VML_ERRMODE_MASK	Specifies mask for error <i>mode</i> selection.

See example below:

Examples

```
mod = vmlgetmode()
accm = IAND(mod, VML_ACCURACY_MASK)
denm = IAND(mod, VML_FTZDAZ_MASK)
errm = IAND(mod, VML_ERRMODE_MASK)

accm = vmlGetMode(void ) & VML_ACCURACY_MASK;
denm = vmlGetMode(void ) & VML_FTZDAZ_MASK;
errm = vmlGetMode(void ) & VML_ERRMODE_MASK;
```

MKLFreeTls

Frees allocated VML/VSL thread local storage memory from within DllMain routine. Use on Windows OS only.*

Syntax

C:

```
void MKLFreeTls( const MKL_UINT fdwReason );
```

Include Files

- C: mkl_vml_functions_win.h

Description

The MKLFreeTls routine frees thread local storage (TLS) memory which has been allocated when using MKL static libraries to link into a DLL on the Windows* OS. The routine should only be used within DllMain.

NOTE

It is only necessary to use MKLFreeTls for TLS data in the VML and VSL domains.

Input Parameters

Name	Type	Description
fdwReason	const MKL_UINT	Reason code from the DllMain call.

Example

```
BOOL WINAPI DllMain(HINSTANCE hInst, DWORD fdwReason, LPVOID lpvReserved)
{
    /*customer code*/
    MKLFreeTls(fdwReason);
    /*customer code*/
}
```

vmlSetErrStatus

Sets the new VML Error Status according to err and stores the previous VML Error Status to olderr.

Syntax

Fortran:

```
olderr = vmlseterrstatus( status )
```

C:

```
olderr = vmlSetErrStatus( status );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>status</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Specifies the VML error status to be set.

Output Parameters

Name	Type	Description
<i>olderr</i>	Fortran: INTEGER C: int	Specifies the former VML error status.

Description

Table "Values of the VML Status" lists possible values of the *err* parameter.

Values of the VML Status

Status	Description
Successful Execution	
VML_STATUS_OK	The execution was completed successfully.
Warnings	
VML_STATUS_ACCURACYWARNING	The execution was completed successfully in a different accuracy mode.
Errors	
VML_STATUS_BADSIZE	The function does not support the preset accuracy mode. The Low Accuracy mode is used instead.
VML_STATUS_BADMEM	NULL pointer is passed.
VML_STATUS_ERRDOM	At least one of array values is out of a range of definition.
VML_STATUS_SING	At least one of the input array values causes a divide-by-zero exception or produces an invalid (QNaN) result.
VML_STATUS_OVERFLOW	An overflow has happened during the calculation process.
VML_STATUS_UNDERFLOW	An underflow has happened during the calculation process.

Examples

```
olderr = vmlSetErrStatus( VML_STATUS_OK );
olderr = vmlSetErrStatus( VML_STATUS_ERRDOM );
olderr = vmlSetErrStatus( VML_STATUS_UNDERFLOW );
```

vmlGetErrStatus

Gets the VML Error Status.

Syntax

Fortran:

```
err = vmlgeterrstatus( )
```

C:

```
err = vmlGetErrStatus( void );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Output Parameters

Name	Type	Description
err	Fortran: INTEGER C: int	Specifies the VML error status.

vmlClearErrStatus

Sets the VML Error Status to VML_STATUS_OK and stores the previous VML Error Status to olderr.

Syntax

Fortran:

```
olderr = vmlclearerrstatus( )
```

C:

```
olderr = vmlClearErrStatus( void );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Output Parameters

Name	Type	Description
olderr	Fortran: INTEGER C: int	Specifies the former VML error status.

vmlSetErrorCallBack

Sets the additional error handler callback function and gets the old callback function.

Syntax

Fortran:

```
oldcallback = vmlseterrorcallback( callback )
```

C:

```
oldcallback = vmlSetErrorCallBack( callback );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Input Parameters

Name

Fortran: *callback* Address of the callback function.

Description

The callback function has the following format:

```
INTEGER FUNCTION ERRFUNC(par)
  TYPE (ERROR_STRUCTURE) par
  ! ...
  ! user error processing
  ! ...
  ! if ERRFUNC= 0 - standard VML error handler
  ! is called after the callback
  ! if ERRFUNC != 0 - standard VML error handler
  ! is not called
END
```

The passed error structure is defined as follows:

```
TYPE ERROR_STRUCTURE SEQUENCE
  INTEGER*4 ICODE
  INTEGER*4 IINDEX
  REAL*8 DBA1
  REAL*8 DBA2
  REAL*8 DBR1
  REAL*8 DBR2
  CHARACTER(64) CFUNCNAME
  INTEGER*4 IFUNCNAMELEN
  REAL*8 DBA1IM
  REAL*8 DBA2IM
  REAL*8 DBR1IM
  REAL*8 DBR2IM
END TYPE ERROR_STRUCTURE
```

C: *callback* Pointer to the callback function.

The callback function has the following format:

```
static int __stdcall
MyHandler(DefVmlErrorHandler*
pContext)
{
    /* Handler body */
}
```

Name**Description**

The passed error structure is defined as follows:

```
typedef struct _DefVmlErrorContext
{
    int iCode; /* Error status value */
    int iIndex; /* Index for bad array
                  element, or bad array
                  dimension, or bad
                  array pointer */

    double dbA1; /* Error argument 1 */
    double dbA2; /* Error argument 2 */
    double dbR1; /* Error result 1 */
    double dbR2; /* Error result 2 */
    char cFuncName[64]; /* Function name */
    int iFuncNameLen; /* Length of functionname*/
    double dbA1Im; /* Error argument 1, imag part*/
    double dbA2Im; /* Error argument 2, imag part*/
    double dbR1Im; /* Error result 1, imag part*/
    double dbR2Im; /* Error result 2, imag part*/
} DefVmlErrorContext;
```

Output Parameters

Name	Type	Description
<i>oldcallback</i>	Fortran 90: INTEGER C: int	Fortran: Address of the former callback function. C: Pointer to the former callback function.

NOTE

This function does not have a FORTRAN 77 interface due to the use of internal structures.

Description

The callback function is called on each VML mathematical function error if `VML_ERRMODE_CALLBACK` error mode is set (see "[Values of the mode Parameter](#)").

Use the `vmlSetErrorHandler()` function if you need to define your own callback function instead of default empty callback function.

The input structure for a callback function contains the following information about the error encountered:

- the input value that caused an error
- location (array index) of this value

- the computed result value
- error code
- name of the function in which the error occurred.

You can insert your own error processing into the callback function. This may include correcting the passed result values in order to pass them back and resume computation. The standard error handler is called after the callback function only if it returns 0.

vmlGetErrorCallBack

Gets the additional error handler callback function.

Syntax

Fortran:

```
callback = vmlgeterrorcallback( )
```

C:

```
callback = vmlGetErrorCallBack( void );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Output Parameters

Name	Description
<i>callback</i>	<p>Fortran 90: Address of the callback function C: Pointer to the callback function</p>

vmlClearErrorCallBack

Deletes the additional error handler callback function and retrieves the former callback function.

Syntax

Fortran:

```
oldcallback = vmlclearerrorcallback( )
```

C:

```
oldcallback = vmlClearErrorCallBack( void );
```

Include Files

- Fortran: mkl_vml.f90
- C: mkl.h

Output Parameters

Name	Type	Description
<i>oldcallback</i>	Fortran 90: INTEGER	Fortran: Address of the former callback function

Name	Type	Description
C:	int	C: Pointer to the former callback function

Statistical Functions

Statistical functions in Intel® MKL are known as the Vector Statistical Library (VSL). It is designed for the purpose of

- generating vectors of pseudorandom, quasi-random, and non-deterministic random numbers
- performing mathematical operations of convolution and correlation
- computing basic statistical estimates for single and double precision multi-dimensional datasets

The corresponding functionality is described in the respective [Random Number Generators](#), [Convolution and Correlation](#), and [Summary Statistics](#) sections.

See VSL performance data in the online VSL Performance Data document available at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>

The basic notion in VSL is a task. The task object is a data structure or descriptor that holds the parameters related to a specific statistical operation: random number generation, convolution and correlation, or summary statistics estimation. Such parameters can be an identifier of a random number generator, its internal state and parameters, data arrays, their shape and dimensions, an identifier of the operation and so forth. You can modify the VSL task parameters using the respective service functionality of the library.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Random Number Generators

Intel MKL VSL provides a set of routines implementing commonly used pseudorandom, quasi-random, or non-deterministic random number generators with continuous and discrete distribution. To improve performance, all these routines were developed using the calls to the highly optimized *Basic Random Number Generators* (BRNGs) and the library of vector mathematical functions (VML, see [Chapter 9, "Vector Mathematical Functions"](#)).

VSL provides interfaces both for Fortran and C languages. For users of the C and C++ languages the `mkl_vsl.h` header file is provided. For users of the Fortran 90 or Fortran 95 language the `mkl_vsl.f90` header file is provided. The `mkl_vsl.fi` header file available in the previous versions of Intel MKL is retained for backward compatibility. For users of the FORTRAN 77 language the `mkl_vsl.f77` header file is provided. All header files are found in the following directory:

```
 ${MKL}/include
```

The `mkl_vsl.f90` header is intended for use with the Fortran `include` clause and is compatible with both standard forms of F90/F95 sources - the free and 72-columns fixed forms. If you need to use the VSL interface with 80- or 132-columns fixed form sources, you may add a new file to your project. That file is formatted as a 72-columns fixed-form source and consists of a single `include` clause as follows:

```
include 'mkl_vsl.f90'
```

This `include` clause causes the compiler to generate the module files `mkl_vsl.mod` and `mkl_vsl_type.mod`, which are used to process the Fortran use clauses referencing to the VSL interface:

```
use mkl_vsl_type
use mkl_vsl
```

Because of this specific feature, you do not need to include the `mkl_vsl.f90` header into each source of your project. You only need to include the header into some of the sources. In any case, make sure that the sources that depend on the VSL interface are compiled after those that include the header so that the module files `mkl_vsl.mod` and `mkl_vsl_type.mod` are generated prior to using them.

The `mkl_vsl.f77` header is intended for use with the Fortran `include` clause as follows:

```
include 'mkl_vsl.f77'
```

NOTE

For Fortran 90 interface, VSL provides both subroutine-style interface and function-style interface. Default interface in this case is a function-style interface. Function-style interface, unlike subroutine-style interface, allows the user to get error status of each routine. Subroutine-style interface is provided for backward compatibility only. To use subroutine-style interface, manually include `mkl_vsl_subroutine.fi` file instead of `mkl_vsl.f90` by changing the line `include 'mkl_vsl.f90'` in `include\mkl.fi` with the line `include 'mkl_vsl_subroutine.fi'`.

For the FORTRAN 77 interface, VSL provides only function-style interface.

All VSL routines can be classified into three major categories:

- Transformation routines for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These routines indirectly call basic random number generators, which are pseudorandom, quasi-random, or non-deterministic random number generators. Detailed description of the generators can be found in [Distribution Generators](#) section.
- Service routines to handle random number streams: create, initialize, delete, copy, save to a binary file, load from a binary file, get the index of a basic generator. The description of these routines can be found in [Service Routines](#) section.
- Registration routines for basic pseudorandom generators and routines that obtain properties of the registered generators (see [Advanced Service Routines](#) section).

The last two categories are referred to as service routines.

Optimization Notice
<p>Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.</p> <p>Notice revision #20110804</p>

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Conventions

This document makes no specific differentiation between random, pseudorandom, and quasi-random numbers, nor between random, pseudorandom, and quasi-random number generators unless the context requires otherwise. For details, refer to the '*Random Numbers*' section in [VSL Notes](#) document provided at the Intel® MKL web page.

All generators of nonuniform distributions, both discrete and continuous, are built on the basis of the uniform distribution generators, called Basic Random Number Generators (BRNGs). The pseudorandom numbers with nonuniform distribution are obtained through an appropriate transformation of the uniformly distributed

pseudorandom numbers. Such transformations are referred to as *generation methods*. For a given distribution, several generation methods can be used. See [VSL Notes](#) for the description of methods available for each generator.

An RNG task determines environment in which random number generation is performed, in particular parameters of the BRNG and its internal state. Output of VSL generators is a stream of random numbers that are used in Monte Carlo simulations. A *random stream descriptor* and a *random stream* are used as synonyms of an *RNG task* in the document unless the context requires otherwise.

The *random stream descriptor* specifies which BRNG should be used in a given transformation method. See the *Random Streams and RNGs in Parallel Computation* section of [VSL Notes](#).

The term *computational node* means a logical or physical unit that can process data in parallel.

Mathematical Notation

The following notation is used throughout the text:

N	The set of natural numbers $N = \{1, 2, 3 \dots\}$.
Z	The set of integers $Z = \{\dots -3, -2, -1, 0, 1, 2, 3 \dots\}$.
R	The set of real numbers.
$\lfloor a \rfloor$	The floor of a (the largest integer less than or equal to a).
\oplus or xor	Bitwise exclusive OR.
C_{α}^{κ} or $\binom{\alpha}{\kappa}$	Binomial coefficient or combination ($\alpha \in R$, $\alpha \geq 0$; $\kappa \in N \cup \{0\}$).
$C_{\alpha}^0 = 1$	
	For $\alpha \geq k$ binomial coefficient is defined as
	$C_{\alpha}^{\kappa} = \frac{\alpha(\alpha - 1) \dots (\alpha - \kappa + 1)}{\kappa!}$
	If $\alpha < k$, then
	$C_{\alpha}^{\kappa} = 0$
$\Phi(x)$	Cumulative Gaussian distribution function
	$\Phi(x) = \int_{-\infty}^x \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right) dy$
	defined over $-\infty < x < +\infty$.
	$\Phi(-\infty) = 0$, $\Phi(+\infty) = 1$.
$\Gamma(\alpha)$	The complete gamma function

$$\Gamma(\alpha) = \int_0^\infty t^{\alpha-1} e^{-t} dt$$

where $\alpha > 0$.

$B(p, q)$

The complete beta function

$$B(p, q) = \int_0^1 t^{p-1} (1-t)^{q-1} dt$$

where $p > 0$ and $q > 0$.

$\text{LCG}(a, c, m)$

Linear Congruential Generator $x_{n+1} = (ax_n + c) \bmod m$, where a is called the *multiplier*, c is called the *increment*, and m is called the *modulus* of the generator.

$\text{MCG}(a, m)$

Multiplicative Congruential Generator $x_{n+1} = (ax_n) \bmod m$ is a special case of Linear Congruential Generator, where the increment c is taken to be 0.

$\text{GFSR}(p, q)$

Generalized Feedback Shift Register Generator

$$x_n = x_{n-p} \oplus x_{n-q}.$$

Naming Conventions

The names of the Fortran routines in VSL random number generators are lowercase (`virnguniform`). The names are not case-sensitive.

The names of the C routines, types, and constants in VSL random number generators are case-sensitive and can contain lowercase and uppercase characters (`viRngUniform`).

The names of generator routines have the following structure:

`v<type of result>rng<distribution>` for the Fortran interface

`v<type of result>Rng<distribution>` for the C interface,

where

- `v` is the prefix of a VSL vector function.
- `<type of result>` is either `s`, `d`, or `i` and specifies one of the following types:

`s` REAL for the Fortran interface

 float for the C interface

`d` DOUBLE PRECISION for the Fortran interface

 double for the C interface

`i` INTEGER for the Fortran interface

 int for the C interface

Prefixes `s` and `d` apply to continuous distributions only, prefix `i` applies only to discrete case.

- `rng` indicates that the routine is a random generator.
- `<distribution>` specifies the type of statistical distribution.

Names of service routines follow the template below:

```
vsl<name>
```

where

- `vsl` is the prefix of a VSL service function.
- `<name>` contains a short function name.

For a more detailed description of service routines, refer to [Service Routines](#) and [Advanced Service Routines](#) sections.

The prototype of each generator routine corresponding to a given probability distribution fits the following structure:

```
status = <function name>( method, stream, n, r, [<distribution parameters>] )
```

where

- `method` defines the method of generation. A detailed description of this parameter can be found in table "[Values of <method> in method parameter](#)". See the next page, where the structure of the `method` parameter name is explained.
- `stream` defines the descriptor of the random stream and must have a non-zero value. Random streams, descriptors, and their usage are discussed further in [Random Streams](#) and [Service Routines](#).
- `n` defines the number of random values to be generated. If `n` is less than or equal to zero, no values are generated. Furthermore, if `n` is negative, an error condition is set.
- `r` defines the destination array for the generated numbers. The dimension of the array must be large enough to store at least `n` random numbers.
- `status` defines the error status of a VSL routine. See [Error Reporting](#) section for a detailed description of error status values.

Additional parameters included into `<distribution parameters>` field are individual for each generator routine and are described in detail in [Distribution Generators](#) section.

To invoke a distribution generator, use a call to the respective VSL routine. For example, to obtain a vector `r`, composed of `n` independent and identically distributed random numbers with normal (Gaussian) distribution, that have the mean value `a` and standard deviation `sigma`, write the following:

for the Fortran interface

```
status = vsrnggaussian( method, stream, n, r, a, sigma )
```

for the C interface

```
status = vsRngGaussian( method, stream, n, r, a, sigma )
```

The name of a `method` parameter has the following structure:

```
VSL_RNG_METHOD_<distribution>_<method>
```

```
VSL_RNG_METHOD_<distribution>_<method>_ACCURATE
```

where

- `<distribution>` is the probability distribution.
- `<method>` is the method name.

Type of the name structure for the `method` parameter corresponds to fast and accurate modes of random number generation (see "[Distribution Generators](#)" section and [VSL Notes](#) for details).

Method names `VSL_RNG_METHOD_<distribution>_<method>`

and

```
VSL_RNG_METHOD_<distribution>_<method>_ACCURATE
```

should be used with

`v<precision>Rng<distribution>`

function only, where

- `<precision>` is
 - `s` for single precision continuous distribution
 - `d` for double precision continuous distribution
 - `i` for discrete distribution
- `<distribution>` is the probability distribution.

is the probability distribution. [Table "Values of <method> in method parameter"](#) provides specific predefined values of the `method` name. The third column contains names of the functions that use the given method.

[Values of <method> in method parameter](#)

Method	Short Description	Functions
STD	Standard method. Currently there is only one method for these functions.	Uniform (continuous) , Uniform (discrete) , UniformBits , UniformBits32 , UniformBits64
BOXMULLER	BOXMULLER generates normally distributed random number x thru the pair of uniformly distributed numbers u_1 and u_2 according to the formula:	Gaussian , GaussianMV
	$x = \sqrt{-2 \ln u_1} \sin 2\pi u_2$	
BOXMULLER2	BOXMULLER2 generates normally distributed random numbers x_1 and x_2 thru the pair of uniformly distributed numbers u_1 and u_2 according to the formulas:	Gaussian , GaussianMV , Lognormal
	$x_1 = \sqrt{-2 \ln u_1} \sin 2\pi u_2$	
	$x_2 = \sqrt{-2 \ln u_1} \cos 2\pi u_2$	
ICDF	Inverse cumulative distribution function method.	Exponential , Laplace , Weibull , Cauchy , Rayleigh , Gumbel , Bernoulli , Geometric , Gaussian , GaussianMV , Lognormal
GNORM	For $\alpha > 1$, a gamma distributed random number is generated as a cube of properly scaled normal random number; for $0.6 \leq \alpha < 1$, a gamma distributed random number is generated using	Gamma

Method	Short Description	Functions
	rejection from Weibull distribution; for $\alpha < 0.6$, a gamma distributed random number is obtained using transformation of exponential power distribution; for $\alpha = 1$, gamma distribution is reduced to exponential distribution.	
CJA	For $\min(p, q) > 1$, Cheng method is used; for $\min(p, q) < 1$, Johnk method is used, if $q + K \cdot p^2 + C \leq 0$ ($K = 0.852\dots$, $C=-0.956\dots$) otherwise, Atkinson switching algorithm is used; for $\max(p, q) < 1$, method of Johnk is used; for $\min(p, q) < 1$, $\max(p, q) > 1$, Atkinson switching algorithm is used (CJA stands for the first letters of Cheng, Johnk, Atkinson); for $p = 1$ or $q = 1$, inverse cumulative distribution function method is used; for $p = 1$ and $q = 1$, beta distribution is reduced to uniform distribution.	Beta
BTPE	Acceptance/rejection method for $n_{trial} \cdot \min(p, 1 - p) \geq 30$ with decomposition into 4 regions: <ul style="list-style-type: none">- 2 parallelograms- triangle- left exponential tail- right exponential tail	Binomial
H2PE	Acceptance/rejection method for large mode of distribution with decomposition into 3 regions: <ul style="list-style-type: none">- rectangular- left exponential tail- right exponential tail	Hypergeometric
PTPE	Acceptance/rejection method for $\lambda \geq 27$ with decomposition into 4 regions: <ul style="list-style-type: none">- 2 parallelograms- triangle- left exponential tail- right exponential tail; otherwise, table lookup method is used.	Poisson
POISNORM	for $\lambda \geq 1$, method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda < 1$, table lookup method is used.	Poisson, PoissonV
NBAR	Acceptance/rejection method for , $\frac{(\alpha - 1) \cdot (1 - p)}{p} \geq 100$ with decomposition into 5 regions: <ul style="list-style-type: none">- rectangular- 2 trapezoid- left exponential tail	NegBinomial

with decomposition into 5 regions:

- rectangular
- 2 trapezoid
- left exponential tail

Method	Short Description	Functions
	- right exponential tail	

NOTE

In this document, routines are often referred to by their base name (`Gaussian`) when this does not lead to ambiguity. In the routine reference, the full name (`vsrnggaussian`, `vsRngGaussian`) is always used in prototypes and code examples.

Basic Generators

VSL provides pseudorandom, quasi-random, and non-deterministic random number generators. This includes the following BRNGs, which differ in speed and other properties:

- the 31-bit multiplicative congruential pseudorandom number generator $MCG(1132489760, 2^{31}-1)$ [[L'Ecuyer99](#)]
- the 32-bit generalized feedback shift register pseudorandom number generator $GFSR(250, 103)$ [[Kirkpatrick81](#)]
- the combined multiple recursive pseudorandom number generator $MRG32k3a$ [[L'Ecuyer99a](#)]
- the 59-bit multiplicative congruential pseudorandom number generator $MCG(13^{13}, 2^{59})$ from NAG Numerical Libraries [[NAG](#)]
- Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [[NAG](#)]
- Mersenne Twister pseudorandom number generator $MT19937$ [[Matsumoto98](#)] with period length $2^{19937}-1$ of the produced sequence
- Set of 6024 Mersenne Twister pseudorandom number generators $MT2203$ [[Matsumoto98](#)], [[Matsumoto00](#)]. Each of them generates a sequence of period length equal to $2^{2203}-1$. Parameters of the generators provide mutual independence of the corresponding sequences.
- SIMD-oriented Fast Mersenne Twister pseudorandom number generator $SFMT19937$ [[Saito08](#)] with a period length equal to $2^{19937}-1$ of the produced sequence.
- Sobol quasi-random number generator [[Sobol76](#)], [[Bratley88](#)], which works in arbitrary dimension. For dimensions greater than 40 the user should supply initialization parameters (initial direction numbers and primitive polynomials or direction numbers) by using `vslNewStreamEx` function. See additional details on interface for registration of the parameters in the library in [VSL Notes](#).
- Niederreiter quasi-random number generator [[Bratley92](#)], which works in arbitrary dimension. For dimensions greater than 318 the user should supply initialization parameters (irreducible polynomials or direction numbers) by using `vslNewStreamEx` function. See additional details on interface for registration of the parameters in the library in [VSL Notes](#).
- Non-deterministic random number generator (RDRAND-based generators only) [[AVX](#)], [[IntelSWMan](#)].

NOTE

You can use a non-deterministic random number generator only if the underlying hardware supports it. For instructions on how to detect if an Intel CPU supports a non-deterministic random number generator see, for example, *Chapter 8: Post-32nm Processor Instructions* in [[AVX](#)] or *Chapter 4: RdRand Instruction Usage* in [[BMT](#)].

NOTE

The time required by some non-deterministic sources to generate a random number is not constant, so you might have to make multiple requests before the next random number is available. VSL limits the number of retries for requests to the non-deterministic source to 10. You can redefine the maximum number of retries during the initialization of the non-deterministic random number generator with the `vslNewStreamEx` function.

For more details on the non-deterministic source implementation for Intel CPUs please refer to Section 7.3.17, Volume 1, *Random Number Generator Instruction* in [IntelSWMan] and Section 4.2.2, *RdRand Retry Loop* in [BMT].

See some testing results for the generators in [VSL Notes](#) and comparative performance data at http://software.intel.com/sites/products/documentation/hpc/mkl/vsl/vsl_data/vsl_performance_data.htm.

VSL provides means of registration of such user-designed generators through the steps described in [Advanced Service Routines](#) section.

For some basic generators, VSL provides two methods of creating independent random streams in multiprocessor computations, which are the leapfrog method and the block-splitting method. These sequence splitting methods are also useful in sequential Monte Carlo.

In addition, MT2203 pseudorandom number generator is a set of 6024 generators designed to create up to 6024 independent random sequences, which might be used in parallel Monte Carlo simulations. Another generator that has the same feature is Wichmann-Hill. It allows creating up to 273 independent random streams. The properties of the generators designed for parallel computations are discussed in detail in [Coddington94].

You may want to design and use your own basic generators. VSL provides means of registration of such user-designed generators through the steps described in [Advanced Service Routines](#) section.

There is also an option to utilize externally generated random numbers in VSL distribution generator routines. For this purpose VSL provides three additional basic random number generators:

- for external random data packed in 32-bit integer array
- for external random data stored in double precision floating-point array; data is supposed to be uniformly distributed over (a, b) interval
- for external random data stored in single precision floating-point array; data is supposed to be uniformly distributed over (a, b) interval.

Such basic generators are called the abstract basic random number generators.

See [VSL Notes](#) for a more detailed description of the generator properties.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

BRNG Parameter Definition

Predefined values for the *brng* input parameter are as follows:

Values of *brng* parameter

Value	Short Description
VSL_BRNG_MCG31	A 31-bit multiplicative congruential generator.
VSL_BRNG_R250	A generalized feedback shift register generator.
VSL_BRNG_MR32K3A	A combined multiple recursive generator with two components of order 3.
VSL_BRNG_MCG59	A 59-bit multiplicative congruential generator.

Value	Short Description
VSL_BRNG_WH	A set of 273 Wichmann-Hill combined multiplicative congruential generators.
VSL_BRNG_MT19937	A Mersenne Twister pseudorandom number generator.
VSL_BRNG_MT2203	A set of 6024 Mersenne Twister pseudorandom number generators.
VSL_BRNG_SFMT19937	A SIMD-oriented Fast Mersenne Twister pseudorandom number generator.
VSL_BRNG_SOBOl	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \leq s \leq 40$; user-defined dimensions are also available.
VSL_BRNG_NIEDERR	A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \leq s \leq 318$; user-defined dimensions are also available.
VSL_BRNG_IABSTRACT	An abstract random number generator for integer arrays.
VSL_BRNG_DABSTRACT	An abstract random number generator for double precision floating-point arrays.
VSL_BRNG_SABSTRACT	An abstract random number generator for single precision floating-point arrays.
VSL_BRNG_NONDETERM	A non-deterministic random number generator.

See [VSL Notes](#) for detailed description.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Random Streams

Random stream (or *stream*) is an abstract source of pseudo- and quasi-random sequences of uniform distribution. You can operate with stream state descriptors only. A stream state descriptor, which holds state descriptive information for a particular BRNG, is a necessary parameter in each routine of a distribution generator. Only the distribution generator routines operate with random streams directly. See [VSL Notes](#) for details.

NOTE

Random streams associated with abstract basic random number generator are called the abstract random streams. See [VSL Notes](#) for detailed description of abstract streams and their use.

You can create unlimited number of random streams by VSL [Service Routines](#) like `NewStream` and utilize them in any distribution generator to get the sequence of numbers of given probability distribution. When they are no longer needed, the streams should be deleted calling service routine `DeleteStream`.

VSL provides service functions `SaveStreamF` and `LoadStreamF` to save random stream descriptive data to a binary file and to read this data from a binary file respectively. See [VSL Notes](#) for detailed description.

Data Types

FORTRAN 77:

```
INTEGER*4 vslstreamstate(2)
```

Fortran 90:

```
TYPE
    VSL_STREAM_STATE
INTEGER*4 descriptor1
INTEGER*4 descriptor2
END
    TYPE VSL_STREAM_STATE
```

C:

```
typedef (void*) VSLStreamStatePtr;
```

See [Advanced Service Routines](#) for the format of the stream state structure for user-designed generators.

Error Reporting

VSL RNG routines return status codes of the performed operation to report errors to the calling program. The application should perform error-related actions and/or recover from the error. The status codes are of integer type and have the following format:

`VSL_ERROR_<ERROR_NAME>` - indicates VSL errors common for all VSL domains.

`VSL_RNG_ERROR_<ERROR_NAME>` - indicates VSL RNG errors.

VSL RNG errors are of negative values while warnings are of positive values. The status code of zero value indicates successful completion of the operation: `VSL_ERROR_OK` (or synonymous `VSL_STATUS_OK`).

Status Codes

Status Code	Description
Common VSL	
<code>VSL_ERROR_OK</code> , <code>VSL_STATUS_OK</code>	No error, execution is successful.
<code>VSL_ERROR_BADARGS</code>	Input argument value is not valid.
<code>VSL_ERROR_CPU_NOT_SUPPORTED</code>	CPU version is not supported.
<code>VSL_ERROR_FEATURE_NOT_IMPLEMENTED</code>	Feature invoked is not implemented.
<code>VSL_ERROR_MEM_FAILURE</code>	System cannot allocate memory.
<code>VSL_ERROR_NULL_PTR</code>	Input pointer argument is NULL.
<code>VSL_ERROR_UNKNOWN</code>	Unknown error.

VSL RNG Specific

<code>VSL_RNG_ERROR_BAD_FILE_FORMAT</code>	File format is unknown.
<code>VSL_RNG_ERROR_BAD_MEM_FORMAT</code>	Descriptive random stream format is unknown.
<code>VSL_RNG_ERROR_BAD_NBITS</code>	The value in NBits field is bad.

Status Code	Description
VSL_RNG_ERROR_BAD_NSEEDS	The value in NSeeds field is bad.
VSL_RNG_ERROR_BAD_STREAM	The random stream is invalid.
VSL_RNG_ERROR_BAD_STREAM_STATE_SIZE	The value in StreamStateSize field is bad.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or >nmax.
VSL_RNG_ERROR_BAD_WORD_SIZE	The value in WordSize field is bad.
VSL_RNG_ERROR_BRNG_NOT_SUPPORTED	BRNG is not supported by the function.
VSL_RNG_ERROR_BRNG_TABLE_FULL	Registration cannot be completed due to lack of free entries in the table of registered BRNGs.
VSL_RNG_ERROR_BRNGS_INCOMPATIBLE	Two BRNGs are not compatible for the operation.
VSL_RNG_ERROR_FILE_CLOSE	Error in closing the file.
VSL_RNG_ERROR_FILE_OPEN	Error in opening the file.
VSL_RNG_ERROR_FILE_READ	Error in reading the file.
VSL_RNG_ERROR_FILE_WRITE	Error in writing the file.
VSL_RNG_ERROR_INVALID_ABSTRACT_STREAM	The abstract random stream is invalid.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is not valid.
VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED	BRNG does not support Leapfrog method.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns zero as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator is exceeded.
VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED	BRNG does not support Skip-Ahead method.
VSL_RNG_ERROR_UNSUPPORTED_FILE_VER	File format version is not supported.
VSL_RNG_ERROR_NONDETERM_NOT_SUPPORTED	Non-deterministic random number generator is not supported on the CPU running the application.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number using non-deterministic random number generator exceeds threshold (see Section 7.2.1.12 <i>Non-deterministic</i> in [VSL Notes] for more details)

VSL RNG Usage Model

A typical algorithm for VSL random number generators is as follows:

1. Create and initialize stream/streams. Functions `vslNewStream`, `vslNewStreamEx`, `vslCopyStream`, `vslCopyStreamState`, `vslLeapfrogStream`, `vslSkipAheadStream`.

-
- 2.** Call one or more RNGs.
 - 3.** Process the output.
 - 4.** Delete the streamstreams. Function `vslDeleteStream`.

NOTE

You may reiterate steps 2-3. Random number streams may be generated for different threads.

The following example demonstrates generation of a random stream that is output of basic generator MT19937. The seed is equal to 777. The stream is used to generate 10,000 normally distributed random numbers in blocks of 1,000 random numbers with parameters $a = 5$ and $\sigma = 2$. Delete the streams after completing the generation. The purpose of the example is to calculate the sample mean for normal distribution with the given parameters.

C Example of VSL RNG Usage

```
#include <stdio.h>
#include "mkl_vsl.h"

int main()
{
    double r[1000]; /* buffer for random numbers */
    double s; /* average */
    VSLStreamStatePtr stream;
    int i, j;

    /* Initializing */
    s = 0.0;
    vslNewStream( &stream, VSL_BRNG_MT19937, 777 );

    /* Generating */
    for ( i=0; i<10; i++ ) {
        vdRngGaussian( VSL_RNG_METHOD_GAUSSIAN_ICDF, stream, 1000, r, 5.0, 2.0 );
        for ( j=0; j<1000; j++ ) {
            s += r[j];
        }
    }
    s /= 10000.0;

    /* Deleting the stream */
    vslDeleteStream( &stream );

    /* Printing results */
    printf( "Sample mean of normal distribution = %f\n", s );
}

return 0;
}
```

Fortran Example of VSL RNG Usage

```
include 'mkl_vsl.f90'

program MKL_VSL_GAUSSIAN

USE MKL_VSL_TYPE
USE MKL_VSL

real(kind=8) r(1000) ! buffer for random numbers
real(kind=8) s         ! average
real(kind=8) a, sigma ! parameters of normal distribution
```

```

TYPE (VSL_STREAM_STATE) :: stream

integer(kind=4) errcode
integer(kind=4) i,j
integer brng,method,seed,n

n = 1000
s = 0.0
a = 5.0
sigma = 2.0
brng=VSL_BRNG_MT19937
method=VSL RNG METHOD GAUSSIAN_ICDF
seed=777

! ***** Initializing *****
errcode=vslnewstream( stream, brng, seed )

! ***** Generating *****
do i = 1,10
    errcode=vdrnggaussian( method, stream, n, r, a, sigma )
    do j = 1, 1000
        s = s + r(j)
    end do
end do

s = s / 10000.0

! ***** Deinitialize *****
errcode=vsldeletestream( stream )

! ***** Printing results *****
print *, "Sample mean of normal distribution = ", s

end

```

Additionally, examples that demonstrate usage of VSL random number generators are available in:

`$(MKL)/examples/vslc/source`
`$(MKL)/examples/vslf/source`

Service Routines

Stream handling comprises routines for creating, deleting, or copying the streams and getting the index of a basic generator. A random stream can also be saved to and then read from a binary file. [Table "Service Routines"](#) lists all available service routines

Service Routines

Routine	Short Description
<code>vslNewStream</code>	Creates and initializes a random stream.
<code>vslNewStreamEx</code>	Creates and initializes a random stream for the generators with multiple initial conditions.
<code>vslI NewAbstractStream</code>	Creates and initializes an abstract random stream for integer arrays.
<code>vslD NewAbstractStream</code>	Creates and initializes an abstract random stream for double precision floating-point arrays.

Routine	Short Description
<code>vslsNewAbstractStream</code>	Creates and initializes an abstract random stream for single precision floating-point arrays.
<code>vslDeleteStream</code>	Deletes previously created stream.
<code>vslCopyStream</code>	Copies a stream to another stream.
<code>vslCopyStreamState</code>	Creates a copy of a random stream state.
<code>vslSaveStreamF</code>	Writes a stream to a binary file.
<code>vslLoadStreamF</code>	Reads a stream from a binary file.
<code>vslSaveStreamM</code>	Writes a random stream descriptive data, including state, to a memory buffer.
<code>vslLoadStreamM</code>	Creates a new stream and reads stream descriptive data, including state, from the memory buffer.
<code>vslGetStreamSize</code>	Computes size of memory necessary to hold the random stream.
<code>vslLeapfrogStream</code>	Initializes the stream by the leapfrog method to generate a subsequence of the original sequence.
<code>vslSkipAheadStream</code>	Initializes the stream by the skip-ahead method.
<code>vslGetStreamStateBrng</code>	Obtains the index of the basic generator responsible for the generation of a given random stream.
<code>vslGetNumRegBrngs</code>	Obtains the number of currently registered basic generators.

Most of the generator-based work comprises three basic steps:

1. Creating and initializing a stream (`vslNewStream`, `vslNewStreamEx`, `vslCopyStream`, `vslCopyStreamState`, `vslLeapfrogStream`, `vslSkipAheadStream`).
2. Generating random numbers with given distribution, see [Distribution Generators](#).
3. Deleting the stream (`vslDeleteStream`).

Note that you can concurrently create multiple streams and obtain random data from one or several generators by using the stream state. You must use the `vslDeleteStream` function to delete all the streams afterwards.

vslNewStream

Creates and initializes a random stream.

Syntax

Fortran:

```
status = vslnewstream( stream, brng, seed )
```

C:

```
status = vslNewStream( &stream, brng, seed );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>brng</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Index of the basic generator to initialize the stream. See Table Values of <i>brng</i> parameter for specific value.
<i>seed</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_UINT	Initial condition of the stream. In the case of a quasi-random number generator seed parameter is used to set the dimension. If the dimension is greater than the dimension that <i>brng</i> can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE(VSL_STREAM_STATE), INTENT(OUT) C: VSLStreamStatePtr*	Stream state descriptor

Description

For a basic generator with number *brng*, this function creates a new stream and initializes it with a 32-bit seed. The seed is an initial value used to select a particular sequence generated by the basic generator *brng*. The function is also applicable for generators with multiple initial conditions. Use this function to create and initialize a new stream with a 32-bit seed only. If you need to provide multiple initial conditions such as several 32-bit or wider seeds, use the function [vslNewStreamEx](#). See [VSL Notes](#) for a more detailed description of stream initialization for different basic generators.

NOTE

This function is not applicable for abstract basic random number generators. Please use [vslNewAbstractStream](#), [vslsNewAbstractStream](#) or [vsldNewAbstractStream](#) to utilize integer, single-precision or double-precision external random data respectively.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Note: deterministic random number generator is not supported.

vslNewStreamEx

Creates and initializes a random stream for generators with multiple initial conditions.

Syntax

Fortran:

```
status = vslnewstreamex( stream, brng, n, params )
```

C:

```
status = vslNewStreamEx( &stream, brng, n, params );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>brng</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Index of the basic generator to initialize the stream. See Table "Values of <i>brng</i> parameter" for specific value.
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Number of initial conditions contained in <i>params</i>
<i>params</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER(KIND=4), INTENT(IN) C: const unsigned int	Array of initial conditions necessary for the basic generator <i>brng</i> to initialize the stream. In the case of a quasi-random number generator only the first element in <i>params</i> parameter is used to set the dimension. If the dimension is greater than the dimension that <i>brng</i> can support or is less than 1, then the dimension is assumed to be equal to 1.

Output Parameters

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE(VSL_STREAM_STATE), INTENT(OUT) C: VSLStreamStatePtr*	Stream state descriptor

Description

The `vslNewStreamEx` function provides an advanced tool to set the initial conditions for a basic generator if its input arguments imply several initialization parameters. Initial values are used to select a particular sequence generated by the basic generator *brng*. Whenever possible, use `vslNewStream`, which is analogous

to `vs1NewStreamEx` except that it takes only one 32-bit initial condition. In particular, `vs1NewStreamEx` may be used to initialize the state table in Generalized Feedback Shift Register Generators (GFSRs). A more detailed description of this issue can be found in [VSL Notes](#).

This function is also used to pass user-defined initialization parameters of quasi-random number generators into the library. See [VSL Notes](#) for the format for their passing and registration in VSL.

NOTE

This function is not applicable for abstract basic random number generators. Please use `vsliNewAbstractStream`, `vs1sNewAbstractStream` or `vs1dNewAbstractStream` to utilize integer, single-precision or double-precision external random data respectively.

Return Values

<code>VSL_ERROR_OK</code> , <code>VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_RNG_ERROR_INVALID_BRNG_INDEX</code>	BRNG index is invalid.
<code>VSL_ERROR_MEM_FAILURE</code>	System cannot allocate memory for <i>stream</i> .
<code>VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED</code>	Note Non-deterministic random number generator is not supported.

vsliNewAbstractStream

Creates and initializes an abstract random stream for integer arrays.

Syntax

Fortran:

```
status = vslineabSTRACTSTREAM( stream, n, ibuf, icallback )
```

C:

```
status = vsliNewAbstractStream( &stream, n, ibuf, icallback );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	<code>FORTRAN 77: INTEGER</code> <code>Fortran 90: INTEGER,</code> INTENT(IN)	Size of the array <i>ibuf</i>
	<code>C: const MKL_INT</code>	
<i>ibuf</i>	<code>FORTRAN 77: INTEGER</code> <code>Fortran 90:</code> INTEGER(KIND=4), INTENT(IN)	Array of <i>n</i> 32-bit integers

Name	Type	Description
	C: const unsigned int	
<i>icallback</i>	See Note below	<p>Fortran: Address of the callback function used for <i>ibuf</i> update</p> <p>C: Pointer to the callback function used for <i>ibuf</i> update</p>

NOTE

Format of the callback function in FORTRAN 77:

```
INTEGER FUNCTION IUPDATEFUNC( stream, n, ibuf, nmin, nmax, idx )
INTEGER*4 stream(2)
INTEGER n
INTEGER*4 ibuf(n)
INTEGER nmin
INTEGER nmax
INTEGER idx
```

Format of the callback function in Fortran 90:

```
INTEGER FUNCTION IUPDATEFUNC[C]( stream, n, ibuf, nmin, nmax, idx )
TYPE(VSL_STREAM_STATE),POINTER :: stream[reference]
INTEGER(KIND=4),INTENT(IN)    :: n[reference]
INTEGER(KIND=4),INTENT(OUT)   :: ibuf[reference](0:n-1)
INTEGER(KIND=4),INTENT(IN)    :: nmin[reference]
INTEGER(KIND=4),INTENT(IN)    :: nmax[reference]
INTEGER(KIND=4),INTENT(IN)    :: idx[reference]
```

Format of the callback function in C:

```
int iUpdateFunc( VSLStreamStatePtr stream, int* n, unsigned int ibuf[], int* nmin, int* nmax,
int* idx );
```

The callback function returns the number of elements in the array actually updated by the function. [Table icallback Callback Function Parameters](#) gives the description of the callback function parameters.

icallback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract random stream descriptor
<i>n</i>	Size of <i>ibuf</i>
<i>ibuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>ibuf</i> to start update $0 \leq idx < n$.

Output Parameters

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2)	Descriptor of the stream state structure

Fortran 90:
 TYPE(VSL_STREAM_STATE),
 INTENT(OUT)

C: VSLStreamStatePtr*

Description

The `vsliNewAbstractStream` function creates a new abstract stream and associates it with an integer array *ibuf* and your callback function *icallback* that is intended for updating of *ibuf* content.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_BADARGS	Parameter <i>n</i> is not positive.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>stream</i> .
VSL_ERROR_NULL_PTR	Either buffer or callback function parameter is a NULL pointer.

vsldNewAbstractStream

Creates and initializes an abstract random stream for double precision floating-point arrays.

Syntax

Fortran:

```
status = vsldnewabstractstream( stream, n, dbuf, a, b, dcallback )
```

C:

```
status = vsldNewAbstractStream( &stream, n, dbuf, a, b, dcallback );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN)	Size of the array <i>dbuf</i>

C: const MKL_INT

Name	Type	Description
<i>dbuf</i>	FORTRAN 77: DOUBLE PRECISION Fortran 90: REAL(KIND=8), INTENT(IN) C: const double	Array of n double precision floating-point random numbers with uniform distribution over interval (a,b)
<i>a</i>	FORTRAN 77: DOUBLE PRECISION Fortran 90: REAL(KIND=8), INTENT(IN) C: const double	Left boundary a
<i>b</i>	FORTRAN 77: DOUBLE PRECISION Fortran 90: REAL(KIND=8), INTENT(IN) C: const double	Right boundary b
<i>dcallback</i>	See Note below	Fortran: Address of the callback function used for update of the array <i>dbuf</i> C: Pointer to the callback function used for update of the array <i>dbuf</i>

Output Parameters

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE(VSL_STREAM_STATE), INTENT(OUT) C: VSLStreamStatePtr*	Descriptor of the stream state structure

NOTE

Format of the callback function in FORTRAN 77:

```
INTEGER FUNCTION DUPDATEFUNC( stream, n, dbuf, nmin, nmax, idx )
INTEGER*4 stream(2)
INTEGER n
DOUBLE PRECISION dbuf(n)
INTEGER nmin
INTEGER nmax
INTEGER idx
```

Format of the callback function in Fortran 90:

```
INTEGER FUNCTION DUPDATEFUNC[C]( stream, n, dbuf, nmin, nmax, idx )
TYPE(VSL_STREAM_STATE),POINTER :: stream[reference]
INTEGER(KIND=4),INTENT(IN) :: n[reference]
REAL(KIND=8),INTENT(OUT) :: dbuf[reference] (0:n-1)
```

```
INTEGER(KIND=4), INTENT(IN)      :: nmin[reference]
INTEGER(KIND=4), INTENT(IN)      :: nmax[reference]
INTEGER(KIND=4), INTENT(IN)      :: idx[reference]
```

Format of the callback function in C:

```
int dUpdateFunc( VSLStreamStatePtr stream, int* n, double dbuf[], int* nmin, int* nmax, int*
idx );
```

The callback function returns the number of elements in the array actually updated by the function. [Table dcallback Callback Function Parameters](#) gives the description of the callback function parameters.

dcallback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract random stream descriptor
<i>n</i>	Size of <i>dbuf</i>
<i>dbuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>dbuf</i> to start update $0 \leq idx < n$.

Description

The `vslsNewAbstractStream` function creates a new abstract stream for double precision floating-point arrays with random numbers of the uniform distribution over interval (a,b) . The function associates the stream with a double precision array *dbuf* and your callback function *dcallback* that is intended for updating of *dbuf* content.

Return Values

<code>VSL_ERROR_OK</code> , <code>VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_BADARGS</code>	Parameter <i>n</i> is not positive.
<code>VSL_ERROR_MEM_FAILURE</code>	System cannot allocate memory for <i>stream</i> .
<code>VSL_ERROR_NULL_PTR</code>	Either buffer or callback function parameter is a NULL pointer.

vslsNewAbstractStream

Creates and initializes an abstract random stream for single precision floating-point arrays.

Syntax

Fortran:

```
status = vslsnewabstractstream( stream, n, sbuf, a, b, scallback )
```

C:

```
status = vslsNewAbstractStream( &stream, n, sbuf, a, b, scallback );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Size of the array <i>sbuf</i>
<i>sbuf</i>	FORTRAN 77: REAL Fortran 90: REAL(KIND=4), INTENT(IN) C: const float	Array of <i>n</i> single precision floating-point random numbers with uniform distribution over interval (<i>a,b</i>)
<i>a</i>	FORTRAN 77: REAL Fortran 90: REAL(KIND=4), INTENT(IN) C: const float	Left boundary <i>a</i>
<i>b</i>	FORTRAN 77: REAL Fortran 90: REAL(KIND=4), INTENT(IN) C: const float	Right boundary <i>b</i>
<i>callback</i>	See Note below	Fortran: Address of the callback function used for update of the array <i>sbuf</i> C: Pointer to the callback function used for update of the array <i>sbuf</i>

Output Parameters

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE(VSL_STREAM_STATE), INTENT(OUT) C: VSLStreamStatePtr*	Descriptor of the stream state structure

NOTE

Format of the callback function in FORTRAN 77:

```
INTEGER FUNCTION SUPDATEFUNC( stream, n, ibuf, nmin, nmax, idx )
INTEGER*4 stream(2)
INTEGER n
REAL sbuf(n)
INTEGER nmin
INTEGER nmax
INTEGER idx
```

Format of the callback function in Fortran 90:

```
INTEGER FUNCTION SUPDATEFUNC[C]( stream, n, sbuf, nmin, nmax, idx )
TYPE(VSL_STREAM_STATE),POINTER :: stream[reference]
INTEGER(KIND=4),INTENT(IN)    :: n[reference]
REAL(KIND=4),   INTENT(OUT)   :: sbuf[reference](0:n-1)
INTEGER(KIND=4),INTENT(IN)    :: nmin[reference]
INTEGER(KIND=4),INTENT(IN)    :: nmax[reference]
INTEGER(KIND=4),INTENT(IN)    :: idx[reference]
```

Format of the callback function in C:

```
int sUpdateFunc( VSLStreamStatePtr stream, int* n, float sbuf[], int* nmin, int* nmax, int* idx );
```

The callback function returns the number of elements in the array actually updated by the function. [Table callback Callback Function Parameters](#) gives the description of the callback function parameters.

callback Callback Function Parameters

Parameters	Short Description
<i>stream</i>	Abstract random stream descriptor
<i>n</i>	Size of <i>sbuf</i>
<i>sbuf</i>	Array of random numbers associated with the stream <i>stream</i>
<i>nmin</i>	Minimal quantity of numbers to update
<i>nmax</i>	Maximal quantity of numbers that can be updated
<i>idx</i>	Position in cyclic buffer <i>sbuf</i> to start update $0 \leq idx < n$.

Description

The `vslsNewAbstractStream` function creates a new abstract stream for single precision floating-point arrays with random numbers of the uniform distribution over interval (a,b) . The function associates the stream with a single precision array *sbuf* and your callback function *callback* that is intended for updating of *sbuf* content.

Return Values

<code>VSL_ERROR_OK</code> , <code>VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_BADARGS</code>	Parameter <i>n</i> is not positive.
<code>VSL_ERROR_MEM_FAILURE</code>	System cannot allocate memory for <i>stream</i> .
<code>VSL_ERROR_NULL_PTR</code>	Either buffer or callback function parameter is a NULL pointer.

vslDeleteStream

Deletes a random stream.

Syntax

Fortran:

```
status = vsldeletestream( stream )
```

C:

```
status = vslDeleteStream( &stream );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input/Output Parameters

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE(VSL_STREAM_STATE), INTENT(OUT) C: VSLStreamStatePtr*	<p>Fortran: Stream state descriptor. Must have non-zero value. After the stream is successfully deleted, the descriptor becomes invalid.</p> <p>C: Stream state descriptor. Must have non-zero value. After the stream is successfully deleted, the pointer is set to NULL.</p>

Description

The function deletes the random stream created by one of the initialization functions.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> parameter is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.

vslCopyStream

Creates a copy of a random stream.

Syntax

Fortran:

```
status = vslcopystream( newstream, srcstream )
```

C:

```
status = vslCopyStream( &newstream, srcstream );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90

- C: mkl.h

Input Parameters

Name	Type	Description
<i>srcstream</i>	FORTRAN 77: INTEGER*4 srcstream(2)	Fortran: Descriptor of the stream to be copied C: Pointer to the stream state structure to be copied

Fortran 90:
 TYPE(VSL_STREAM_STATE),
 INTENT(IN)

C: const VSLStreamStatePtr

Output Parameters

Name	Type	Description
<i>newstream</i>	FORTRAN 77: INTEGER*4 newstream(2)	Copied random stream descriptor

Fortran 90:
 TYPE(VSL_STREAM_STATE),
 INTENT(OUT)

C: VSLStreamStatePtr*

Description

The function creates an exact copy of *srcstream* and stores its descriptor to *newstream*.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>srcstream</i> parameter is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>srcstream</i> is not a valid random stream.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for <i>newstream</i> .

vslCopyStreamState

Creates a copy of a random stream state.

Syntax

Fortran:

```
status = vslcopystreamstate( deststream, srcstream )
```

C:

```
status = vslCopyStreamState( deststream, srcstream );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>srcstream</i>	FORTRAN 77: INTEGER*4 srcstream(2)	Fortran: Descriptor of the destination stream where the state of <i>scrstream</i> stream is copied

Fortran 90:
 TYPE(VSL_STREAM_STATE),
 INTENT(IN)

C: const VSLStreamStatePtr

Output Parameters

Name	Type	Description
<i>deststream</i>	FORTRAN 77: INTEGER*4 deststream(2)	Fortran: Descriptor of the stream with the state to be copied

Fortran 90:
 TYPE(VSL_STREAM_STATE),
 INTENT(OUT)

C: VSLStreamStatePtr

Description

The `vslCopyStreamState` function copies a stream state from *srcstream* to the existing *deststream* stream. Both the streams should be generated by the same basic generator. An error message is generated when the index of the BRNG that produced *deststream* stream differs from the index of the BRNG that generated *srcstream* stream.

Unlike `vslCopyStream` function, which creates a new stream and copies both the stream state and other data from *srcstream*, the function `vslCopyStreamState` copies only *srcstream* stream state data to the generated *deststream* stream.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either <i>srcstream</i> or <i>deststream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	Either <i>srcstream</i> or <i>deststream</i> is not a valid random stream.
VSL_RNG_ERROR_BRNGS_INCOMPATIBLE	BRNG associated with <i>srcstream</i> is not compatible with BRNG associated with <i>deststream</i> .

vslSaveStreamF

Writes random stream descriptive data, including stream state, to binary file.

Syntax

Fortran:

```
errstatus = vslsavestreamf( stream, fname )
```

C:

```
errstatus = vslSaveStreamF( stream, fname );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2)	Random stream to be written to the file
	Fortran 90: TYPE(VSL_STREAM_STATE), INTENT(IN)	
	C: const VSLStreamStatePtr	
<i>fname</i>	FORTRAN 77: CHARACTER(*) Fortran 90: CHARACTER(*), INTENT(IN)	Fortran: File name specified as a C-style null-terminated string C: File name specified as a null-terminated string
	C: const char*	

Output Parameters

Name	Type	Description
<i>errstatus</i>	Fortran: INTEGER C: int	Error status of the operation

Description

The `vslSaveStreamF` function writes the random stream descriptive data, including the stream state, to the binary file. Random stream descriptive data is saved to the binary file with the name *fname*. The random stream *stream* must be a valid stream created by `vslNewStream`-like or `vslCopyStream`-like service routines. If the stream cannot be saved to the file, *errstatus* has a non-zero value. The random stream can be read from the binary file using the `vslLoadStreamF` function.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either <i>fname</i> or <i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_RNG_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_RNG_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.

vslLoadStreamF

Creates new stream and reads stream descriptive data, including stream state, from binary file.

Syntax

Fortran:

```
errstatus = vslloadstreamf( stream, fname )
```

C:

```
errstatus = vslLoadStreamF( &stream, fname );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>fname</i>	FORTRAN 77: CHARACTER(*) Fortran 90: CHARACTER(*), INTENT(IN) C: const char*	Fortran: File name specified as a C-style null-terminated string C: File name specified as a null-terminated string

Output Parameters

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE(VSL_STREAM_STATE), INTENT(OUT) C: VSLStreamStatePtr*	Fortran: Descriptor of a new random stream C: Pointer to a new random stream
<i>errstatus</i>	Fortran: INTEGER C: int	Error status of the operation

Description

The `vslLoadStreamF` function creates a new stream and reads stream descriptive data, including the stream state, from the binary file. A new random stream is created using the stream descriptive data from the binary file with the name *fname*. If the stream cannot be read (for example, an I/O error occurs or the file format is invalid), *errstatus* has a non-zero value. To save random stream to the file, use `vslSaveStreamF` function.

CAUTION

Calling `vslLoadStreamF` with a previously initialized *stream* pointer can have unintended consequences such as a memory leak. To initialize a stream which has been in use until calling `vslLoadStreamF`, you should call the `vslDeleteStream` function first to deallocate the resources.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>fname</i> is a NULL pointer.
VSL_RNG_ERROR_FILE_OPEN	Indicates an error in opening the file.
VSL_RNG_ERROR_FILE_WRITE	Indicates an error in writing the file.
VSL_RNG_ERROR_FILE_CLOSE	Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE	System cannot allocate memory for internal needs.
VSL_RNG_ERROR_BAD_FILE_FORMAT	Unknown file format.
VSL_RNG_ERROR_UNSUPPORTED_FILE_VER	File format version is unsupported.
VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED	Note: Non-deterministic random number generator is not supported.

vslSaveStreamM

Writes random stream descriptive data, including stream state, to a memory buffer.

Syntax

Fortran:

```
errstatus = vslsavestreamm( stream, memptr )
```

C:

```
errstatus = vslSaveStreamM( stream, memptr );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 <i>stream</i> (2)	Random stream to be written to the memory
	Fortran 90: TYPE(VSL_STREAM_STATE), INTENT(IN)	
	C: const VSLStreamStatePtr	
<i>memptr</i>	FORTRAN 77: INTEGER*1 Fortran 90: INTEGER(KIND=1), DIMENSION(*), INTENT(IN)	Fortran: Memory buffer to save random stream descriptive data to C: Memory buffer to save random stream descriptive data to

Name	Type	Description
	C: char*	

Output Parameters

Name	Type	Description
errstatus	Fortran: INTEGER	Error status of the operation
	C: int	

Description

The `vslSaveStreamM` function writes the random stream descriptive data, including the stream state, to the memory at `memptr`. Random stream `stream` must be a valid stream created by `vslNewStream`-like or `vslCopyStream`-like service routines. The `memptr` parameter must be a valid pointer to the memory of size sufficient to hold the random stream `stream`. Use the service routine `vslGetStreamSize` to determine this amount of memory.

If the stream cannot be saved to the memory, `errstatus` has a non-zero value. The random stream can be read from the memory pointed by `memptr` using the `vslLoadStreamM` function.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	Either <code>memptr</code> or <code>stream</code> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<code>stream</code> is a NULL pointer.

vslLoadStreamM

Creates a new stream and reads stream descriptive data, including stream state, from the memory buffer.

Syntax

Fortran:

```
errstatus = vslloadstreamm( stream, memptr )
```

C:

```
errstatus = vslLoadStreamM( &stream, memptr );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<code>memptr</code>	FORTRAN 77: INTEGER*1	Fortran: Memory buffer to load random stream descriptive data from

Name	Type	Description
	<pre>Fortran 90: INTEGER(KIND=1), DIMENSION(*), INTENT(IN) C: const char*</pre>	C: Memory buffer to load random stream descriptive data from

Output Parameters

Name	Type	Description
<i>stream</i>	<pre>FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE(VSL_STREAM_STATE), INTENT(OUT) C: VSLStreamStatePtr*</pre>	Fortran: Descriptor of a new random stream C: Pointer to a new random stream
<i>errstatus</i>	<pre>Fortran: INTEGER C: int</pre>	Error status of the operation

Description

The `vslLoadStreamM` function creates a new stream and reads stream descriptive data, including the stream state, from the memory buffer. A new random stream is created using the stream descriptive data from the memory pointer by `memptr`. If the stream cannot be read (for example, `memptr` is invalid), `errstatus` has a non-zero value. To save random stream to the memory, use `vslSaveStreamM` function. Use the service routine `vslGetStreamSize` to determine the amount of memory sufficient to hold the random stream.

CAUTION

Calling `LoadStreamM` with a previously initialized `stream` pointer can have unintended consequences such as a memory leak. To initialize a stream which has been in use until calling `vslLoadStreamM`, you should call the `vslDeleteStream` function first to deallocate the resources.

Return Values

<code>VSL_ERROR_OK</code> , <code>VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<code>memptr</code> is a NULL pointer.
<code>VSL_ERROR_MEM_FAILURE</code>	System cannot allocate memory for internal needs.
<code>VSL_RNG_ERROR_BAD_MEM_FORMAT</code>	Descriptive random stream format is unknown.
<code>VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED</code>	Non-deterministic random number generator is not supported.

`vslGetSize`

Computes size of memory necessary to hold the random stream.

Syntax

Fortran:

```
memsize = vslgetstreamsize( stream )
```

C:

```
memsize = vslGetStreamSize( stream );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2)	Random stream

Fortran 90:
 TYPE(VSL_STREAM_STATE),
 INTENT(IN)

C: const VSLStreamStatePtr

Output Parameters

Name	Type	Description
<i>memsize</i>	Fortran: INTEGER C: int	Amount of memory in bytes necessary to hold descriptive data of random stream <i>stream</i>

Description

The `vslGetStreamSize` function returns the size of memory in bytes which is necessary to hold the given random stream. Use the output of the function to allocate the buffer to which you will save the random stream by means of the `vslSaveStreamM` function.

Return Values

<code>VSL_ERROR_OK</code> , <code>VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>stream</i> is a NULL pointer.

`vslLeapfrogStream`

Initializes a stream using the leapfrog method.

Syntax

Fortran:

```
status = vslleapfrogstream( stream, k, nstreams )
```

C:

```
status = vslLeapfrogStream( stream, k, nstreams );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

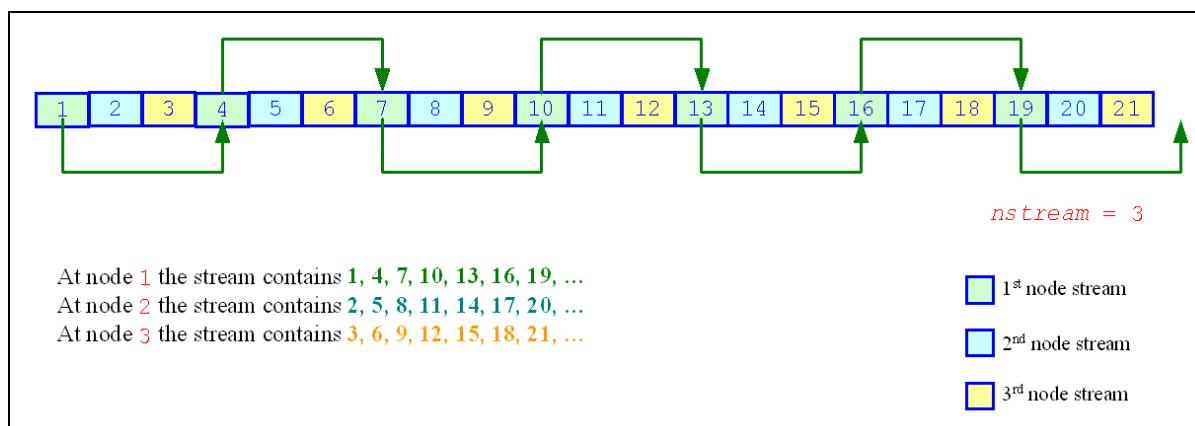
Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2)	Fortran: Descriptor of the stream to which leapfrog method is applied
	Fortran 90: TYPE(VSL_STREAM_STATE), INTENT(IN)	C: Pointer to the stream state structure to which leapfrog method is applied
	C: VSLStreamStatePtr	
<i>k</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN)	Index of the computational node, or stream number
	C: const MKL_INT	
<i>nstreams</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN)	Largest number of computational nodes, or stride
	C: const MKL_INT	

Description

The `vslLeapfrogStream` function generates random numbers in a random stream with non-unit stride. This feature is particularly useful in distributing random numbers from the original stream across the *nstreams* buffers without generating the original random sequence with subsequent manual distribution.

One of the important applications of the leapfrog method is splitting the original sequence into non-overlapping subsequences across *nstreams* computational nodes. The function initializes the original random stream (see [Figure "Leapfrog Method"](#)) to generate random numbers for the computational node *k*, $0 \leq k < nstreams$, where *nstreams* is the largest number of computational nodes used.

Leapfrog Method



The leapfrog method is supported only for those basic generators that allow splitting elements by the leapfrog method, which is more efficient than simply generating them by a generator with subsequent manual distribution across computational nodes. See [VSL Notes](#) for details.

For quasi-random basic generators, the leapfrog method allows generating individual components of quasi-random vectors instead of whole quasi-random vectors. In this case *nstreams* parameter should be equal to the dimension of the quasi-random vector while *k* parameter should be the index of a component to be generated ($0 \leq k < nstreams$). Other parameters values are not allowed.

The following code illustrates the initialization of three independent streams using the leapfrog method:

Fortran 90 Code for Leapfrog Method

```
...
TYPE(VSL_STREAM_STATE)    ::stream1
TYPE(VSL_STREAM_STATE)    ::stream2
TYPE(VSL_STREAM_STATE)    ::stream3

! Creating 3 identical streams
status = vslnewstream(stream1, VSL_BRNG_MCG31, 174)
status = vslcopystream(stream2, stream1)
status = vslcopystream(stream3, stream1)

! Leapfrogging the streams
status = vslleapfrogstream(stream1, 0, 3)
status = vslleapfrogstream(stream2, 1, 3)
status = vslleapfrogstream(stream3, 2, 3)

! Generating random numbers
...
! Deleting the streams
status = vsldeletestream(stream1)
status = vsldeletestream(stream2)
status = vsldeletestream(stream3)
...
```

C Code for Leapfrog Method

```
...
VSLStreamStatePtr stream1;
VSLStreamStatePtr stream2;
VSLStreamStatePtr stream3;

/* Creating 3 identical streams */
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);
status = vslCopyStream(&stream2, stream1);
status = vslCopyStream(&stream3, stream1);

/* Leapfrogging the streams
*/
status = vsllLeapfrogStream(stream1, 0, 3);
status = vsllLeapfrogStream(stream2, 1, 3);
status = vsllLeapfrogStream(stream3, 2, 3);

/* Generating random numbers
*/
...
/* Deleting the streams
*/
status = vslDeleteStream(&stream1);
```

```
status = vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
...
```

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED	BRNG does not support Leapfrog method.

vslSkipAheadStream

Initializes a stream using the block-splitting method.

Syntax

Fortran:

```
status = vslskipaheadstream( stream, nskip )
```

C:

```
status = vslSkipAheadStream( stream, nskip );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

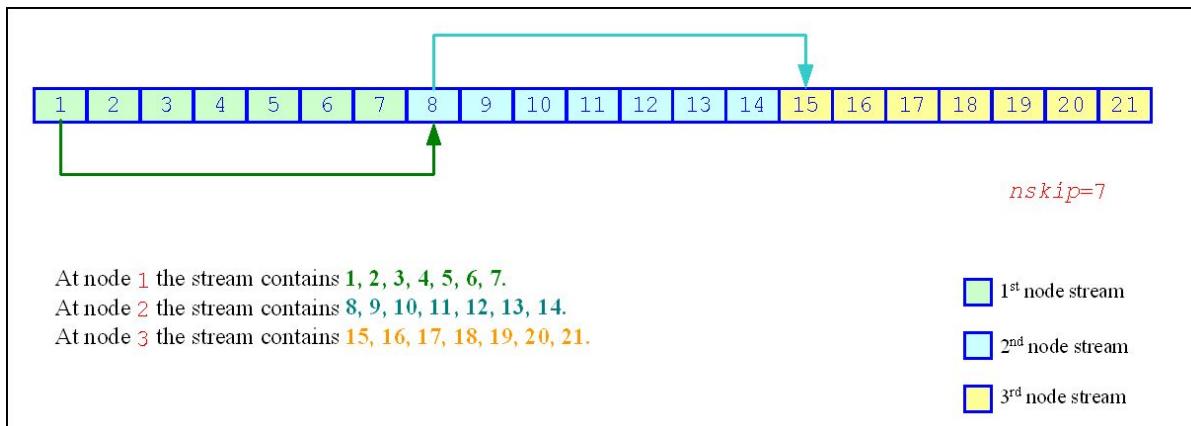
Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2)	Fortran: Descriptor of the stream to which block-splitting method is applied
	Fortran 90: TYPE(VSL_STREAM_STATE), INTENT(IN)	C: Pointer to the stream state structure to which block-splitting method is applied
	C: VSLStreamStatePtr	
<i>nskip</i>	FORTRAN 77: INTEGER*4 nskip(2)	Number of skipped elements
	Fortran 90: INTEGER(KIND=8), INTENT(IN)	
	C: const long long int	

Description

The `vslSkipAheadStream` function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is *nskip*, then the original random sequence may be split by `vslSkipAheadStream` into non-overlapping blocks of

nskip size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method. (see Figure "Block-Splitting Method").

Block-Splitting Method



The skip-ahead method is supported only for those basic generators that allow skipping elements by the skip-ahead method, which is more efficient than simply generating them by generator with subsequent manual skipping. See [VSL Notes](#) for details.

Please note that for quasi-random basic generators the skip-ahead method works with components of quasi-random vectors rather than with whole quasi-random vectors. Therefore, to skip *NS* quasi-random vectors, set the *nskip* parameter equal to the *NS*DIMEN*, where *DIMEN* is the dimension of the quasi-random vector. If this operation results in exceeding the period of the quasi-random number generator, which is $2^{32}-1$, the library returns the `VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED` error code.

The following code illustrates how to initialize three independent streams using the `vslSkipAheadStream` function:

Fortran 90 Code for Block-Splitting Method

```
...
type(VSL_STREAM_STATE) ::stream1
type(VSL_STREAM_STATE) ::stream2
type(VSL_STREAM_STATE) ::stream3

! Creating the 1st stream
status = vslnewstream(stream1, VSL_BRNG_MCG31, 174)

! Skipping ahead by 7 elements the 2nd stream
status = vslcopystream(stream2, stream1);
status = vslskipaheadstream(stream2, 7);

! Skipping ahead by 7 elements the 3rd stream
status = vslcopystream(stream3, stream2);
status = vslskipaheadstream(stream3, 7);

! Generating random numbers
...
! Deleting the streams
status = vsdeletestream(stream1)
```

```
status = vslDeleteStream(stream2)
status = vslDeleteStream(stream3)
...
```

C Code for Block-Splitting Method

```
VSLStreamStatePtr stream1;
VSLStreamStatePtr stream2;
VSLStreamStatePtr stream3;

/* Creating the 1st stream
 */
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);

/* Skipping ahead by 7 elements the 2nd stream */
status = vslCopyStream(&stream2, stream1);
status = vslSkipAheadStream(stream2, 7);

/* Skipping ahead by 7 elements the 3rd stream */
status = vslCopyStream(&stream3, stream2);
status = vslSkipAheadStream(stream3, 7);

/* Generating random numbers
 */
...
/* Deleting the streams
 */
status = vslDeleteStream(&stream1);
status = vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
...
```

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED	BRNG does not support the Skip-Ahead method.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the quasi-random number generator is exceeded.

vslGetStreamStateBrng

Returns index of a basic generator used for generation of a given random stream.

Syntax

Fortran:

```
brng = vslgetstreamstatebrng( stream )
```

C:

```
brng = vslGetStreamStateBrng( stream );
```

Include Files

- Fortran: mkl.fi

- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2)	Fortran: Descriptor of the stream state C: Pointer to the stream state structure

Fortran 90:
 TYPE(VSL_STREAM_STATE),
 TINTENT(IN)
C: const VSLStreamStatePtr

Output Parameters

Name	Type	Description
<i>brng</i>	Fortran: INTEGER C: int	Index of the basic generator assigned for the generation of <i>stream</i> ; negative in case of an error

Description

The vslGetStreamStateBrng function retrieves the index of a basic generator used for generation of a given random stream.

Return Values

VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.

vslGetNumRegBrngs

Obtains the number of currently registered basic generators.

Syntax

Fortran:

```
nregbrngs = vslgetnumregbrngs( )
```

C:

```
nregbrngs = vslGetNumRegBrngs( void );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Output Parameters

Name	Type	Description
<code>nregbrngs</code>	Fortran: INTEGER C: int	Number of basic generators registered at the moment of the function call

Description

The `vslGetNumRegBrngs` function obtains the number of currently registered basic generators. Whenever user registers a user-designed basic generator, the number of registered basic generators is incremented. The maximum number of basic generators that can be registered is determined by the `VSL_MAX_REG_BRNGS` parameter.

Distribution Generators

Intel MKL VSL routines are used to generate random numbers with different types of distribution. Each function group is introduced below by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence for both Fortran and C-interface and the explanation of input and output parameters. [Table "Continuous Distribution Generators"](#) and [Table "Discrete Distribution Generators"](#) list the random number generator routines with data types and output distributions, and sets correspondence between data types of the generator routines and the basic random number generators.

Continuous Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
<code>vRngUniform</code>	s, d	s, d	Uniform continuous distribution on the interval $[a,b)$
<code>vRngGaussian</code>	s, d	s, d	Normal (Gaussian) distribution
<code>vRngGaussianMV</code>	s, d	s, d	Multivariate normal (Gaussian) distribution
<code>vRngExponential</code>	s, d	s, d	Exponential distribution
<code>vRngLaplace</code>	s, d	s, d	Laplace distribution (double exponential distribution)
<code>vRngWeibull</code>	s, d	s, d	Weibull distribution
<code>vRngCauchy</code>	s, d	s, d	Cauchy distribution
<code>vRngRayleigh</code>	s, d	s, d	Rayleigh distribution
<code>vRngLognormal</code>	s, d	s, d	Lognormal distribution
<code>vRngGumbel</code>	s, d	s, d	Gumbel (extreme value) distribution
<code>vRngGamma</code>	s, d	s, d	Gamma distribution
<code>vRngBeta</code>	s, d	s, d	Beta distribution

Discrete Distribution Generators

Type of Distribution	Data Types	BRNG Data Type	Description
<code>vRngUniform</code>	i	d	Uniform discrete distribution on the interval $[a,b)$

Type of Distribution	Data Types	BRNG Data Type	Description
vRngUniformBits	i	i	Underlying BRNG integer recurrence
vRngUniformBits32	i	i	Uniformly distributed bits in 32-bit chunks
vRngUniformBits64	i	i	Uniformly distributed bits in 64-bit chunks
vRngBernoulli	i	s	Bernoulli distribution
vRngGeometric	i	s	Geometric distribution
vRngBinomial	i	d	Binomial distribution
vRngHypergeometric	i	d	Hypergeometric distribution
vRngPoisson	i	s (for VSL_RNG_METHOD_POISSON_POISNORM) s (for distribution parameter $\lambda \geq 27$) and d (for $\lambda < 27$) (for VSL_RNG_METHOD_POISSON_PTPE)	Poisson distribution
vRngPoissonV	i	s	Poisson distribution with varying mean
vRngNegBinomial	i	d	Negative binomial distribution, or Pascal distribution

Modes of random number generation

The library provides two modes of random number generation, accurate and fast. Accurate generation mode is intended for the applications that are highly demanding to accuracy of calculations. When used in this mode, the generators produce random numbers lying completely within definitional domain for all values of the distribution parameters. For example, random numbers obtained from the generator of continuous distribution that is uniform on interval $[a,b]$ belong to this interval irrespective of what a and b values may be. Fast mode provides high performance of generation and also guarantees that generated random numbers belong to the definitional domain except for some specific values of distribution parameters. The generation mode is set by specifying relevant value of the method parameter in generator routines. List of distributions that support accurate mode of generation is given in the table below.

Distribution Generators Supporting Accurate Mode

Type of Distribution	Data Types
vRngUniform	s, d
vRngExponential	s, d
vRngWeibull	s, d
vRngRayleigh	s, d
vRngLognormal	s, d
vRngGamma	s, d
vRngBeta	s, d

See additional details about accurate and fast mode of random number generation in [VSL Notes](#).

New method names

The current version of Intel MKL has a modified structure of VSL RNG method names. (See [RNG Naming Conventions](#) for details.) The old names are kept for backward compatibility. The set correspondence between the new and legacy method names for VSL random number generators.

Method Names for Continuous Distribution Generators

RNG	Legacy Method Name	New Method Name
vRngUniform	VSL_METHOD_SUNIFORM_STD, VSL_METHOD_DUNIFORM_STD, VSL_METHOD_SUNIFORM_STD_ACCURATE, VSL_METHOD_DUNIFORM_STD_ACCURATE	VSL_RNG_METHOD_UNIFORM_STD, VSL_RNG_METHOD_UNIFORM_STD_ACCURATE
vRngGaussian	VSL_METHOD_SGAUSSIAN_BOXMULLER, VSL_METHOD_SGAUSSIAN_BOXMULLER2, VSL_METHOD_SGAUSSIAN_ICDF, VSL_METHOD_DGAUSSIAN_BOXMULLER, VSL_METHOD_DGAUSSIAN_BOXMULLER2, VSL_METHOD_DGAUSSIAN_ICDF	VSL_RNG_METHOD_GAUSSIAN_BOXMULLER, VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2, VSL_RNG_METHOD_GAUSSIAN_ICDF
vRngGaussianMV	VSL_METHOD_SGAUSSIANMV_BOXMULLER, VSL_METHOD_SGAUSSIANMV_BOXMULLER2, VSL_METHOD_SGAUSSIANMV_ICDF, VSL_METHOD_DGAUSSIANMV_BOXMULLER, VSL_METHOD_DGAUSSIANMV_BOXMULLER2, VSL_METHOD_DGAUSSIANMV_ICDF	VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER , VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2 , VSL_RNG_METHOD_GAUSSIANMV_ICDF
vRngExponential	VSL_METHOD_SEXPONENTIAL_ICDF, VSL_METHOD_DEXPOENTIAL_ICDF, VSL_METHOD_SEXPONENTIAL_ICDF_ACCURATE ,	VSL_RNG_METHOD_EXPONENTIAL_ICDF, VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE
vRngLaplace	VSL_METHOD_SLAPLACE_ICDF, VSL_METHOD_DLAPLACE_ICDF	VSL_RNG_METHOD_LAPLACE_ICDF
vRngWeibull	VSL_METHOD_SWEIFULL_ICDF, VSL_METHOD_DWEIBULL_ICDF, VSL_METHOD_SWEIFULL_ICDF_ACCURATE, VSL_METHOD_DWEIBULL_ICDF_ACCURATE	VSL_RNG_METHOD_WEIBULL_ICDF, VSL_RNG_METHOD_WEIBULL_ICDF_ACCURATE
vRngCauchy	VSL_METHOD_SCAUCHY_ICDF, VSL_METHOD_DCAUCHY_ICDF	VSL_RNG_METHOD_CAUCHY_ICDF
vRngRayleigh	VSL_METHOD_SRAYLEIGH_ICDF, VSL_METHOD_DRAYLEIGH_ICDF, VSL_METHOD_SRAYLEIGH_ICDF_ACCURATE, VSL_METHOD_DRAYLEIGH_ICDF_ACCURATE	VSL_RNG_METHOD_RAYLEIGH_ICDF, VSL_RNG_METHOD_RAYLEIGH_ICDF_ACCURATE
vRngLognormal	VSL_METHOD_SLOGNORMAL_BOXMULLER2, VSL_METHOD_DLGNORMAL_BOXMULLER2, VSL_METHOD_SLOGNORMAL_BOXMULLER2_ACCURATE ,	VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2 ,
		VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2_ACCURATE
	VSL_METHOD_SLOGNORMAL_ICDF, VSL_METHOD_DLGNORMAL_ICDF, VSL_METHOD_SLOGNORMAL_ICDF_ACCURATE ,	VSL_RNG_METHOD_LOGNORMAL_ICDF, VSL_RNG_METHOD_LOGNORMAL_ICDF_ACCURATE
	VSL_METHOD_DLGNORMAL_ICDF_ACCURATE	

RNG	Legacy Method Name	New Method Name
vRngGumbel	VSL_METHOD_SGUMBEL_ICDF, VSL_METHOD_DGUMBEL_ICDF	VSL_RNG_METHOD_GUMBEL_ICDF
vRngGamma	VSL_METHOD_SGAMMA_GNORM, VSL_METHOD_DGAMMA_GNORM, VSL_METHOD_SGAMMA_GNORM_ACCURATE, VSL_METHOD_DGAMMA_GNORM_ACCURATE	VSL_RNG_METHOD_GAMMA_GNORM, VSL_RNG_METHOD_GAMMA_GNORM_ACCURATE
vRngBeta	VSL_METHOD_SBETA_CJA, VSL_METHOD_DBETA_CJA, VSL_METHOD_SBETA_CJA_ACCURATE, VSL_METHOD_DBETA_CJA_ACCURATE	VSL_RNG_METHOD_BETA_CJA, VSL_RNG_METHOD_BETA_CJA_ACCURATE

Method Names for Discrete Distribution Generators

RNG	Legacy Method Name	New Method Name
vRngUniform	VSL_METHOD_IUNIFORM_STD	VSL_RNG_METHOD_UNIFORM_STD
vRngUniformBits	VSL_METHOD_IUNIFORMBITS_STD	VSL_RNG_METHOD_UNIFORMBITS_STD
vRngBernoulli	VSL_METHOD_IBERNOULLI_ICDF	VSL_RNG_METHOD_BERNOULLI_ICDF
vRngGeometric	VSL_METHOD_IGEOMETRIC_ICDF	VSL_RNG_METHOD_GEOMETRIC_ICDF
vRngBinomial	VSL_METHOD_IBINOMIAL_BTPE	VSL_RNG_METHOD_BINOMIAL_BTPE
vRngHypergeometric	VSL_METHOD_IHYPERGEOMETRIC_H2PE	VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE
vRngPoisson	VSL_METHOD_IPOISSON_PTPE, VSL_METHOD_IPOISSON_POISNORM	VSL_RNG_METHOD_POISSON_PTPE, VSL_RNG_METHOD_POISSON_POISNORM
vRngPoissonV	VSL_METHOD_IPOISSONV_POISNORM	VSL_RNG_METHOD_POISSONV_POISNORM
vRngNegBinomial	VSL_METHOD_INEBINOMIAL_NBAR	VSL_RNG_METHOD_NEGBINOMIAL_NBAR

Continuous Distributions

This section describes routines for generating random numbers with continuous distribution.

vRngUniform

Generates random numbers with uniform distribution.

Syntax

Fortran:

```
status = vsrnguniform( method, stream, n, r, a, b )
status = vdrguniform( method, stream, n, r, a, b )
```

C:

```
status = vsRngUniform( method, stream, n, r, a, b );
status = vdRngUniform( method, stream, n, r, a, b );
```

Include Files

- Fortran: mkl.fi

- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT</pre>	Generation method; the specific values are as follows: <pre>VSL_RNG_METHOD_UNIFORM_STD VSL_RNG_METHOD_UNIFORM_STD_ACCURATE</pre>
<i>stream</i>	<pre>FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr</pre>	Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT</pre>	Number of random values to be generated
<i>a</i>	<pre>FORTRAN 77: REAL for vsrnguniform DOUBLE PRECISION for vdrnguniform Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnguniform REAL(KIND=8), INTENT(IN) for vdrnguniform C: const float for vsRngUniform const double for vdRngUniform</pre>	Left bound <i>a</i>
<i>b</i>	<pre>FORTRAN 77: REAL for vsrnguniform DOUBLE PRECISION for vdrnguniform Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnguniform REAL(KIND=8), INTENT(IN) for vdrnguniform C: const float for vsRngUniform</pre>	Right bound <i>b</i>

Name	Type	Description
	const double for vdRngUniform	

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: REAL for vsrnguniform DOUBLE PRECISION for vdrnguniform Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrnguniform REAL(KIND=8), INTENT(OUT) for vdrnguniform C: float* for vsRngUniform double* for vdRngUniform	Vector of <i>n</i> random numbers uniformly distributed over the interval $[a, b]$

Description

The `vRngUniform` function generates random numbers uniformly distributed over the interval $[a, b]$, where a, b are the left and right bounds of the interval, respectively, and $a, b \in R ; a < b$.

The probability density function is given by:

$$f(x) = \frac{1}{b-a} \quad a \leq x \leq b$$

The cumulative distribution function is as follows:

$$F(x) = \frac{x-a}{b-a} \quad a \leq x \leq b$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngGaussian

Generates normally distributed random numbers.

Syntax

Fortran:

```
status = vsrnggaussian( method, stream, n, r, a, sigma )
status = vdrnggaussian( method, stream, n, r, a, sigma )
```

C:

```
status = vsRngGaussian( method, stream, n, r, a, sigma );
status = vdRngGaussian( method, stream, n, r, a, sigma );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
method	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Generation method. The specific values are as follows: <div style="background-color: #f0f0f0; padding: 2px;">VSL_RNG_METHOD_GAUSSIAN_BOXMULLER</div> <div style="background-color: #f0f0f0; padding: 2px;">VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2</div> <div style="background-color: #f0f0f0; padding: 2px;">VSL_RNG_METHOD_GAUSSIAN_ICDF</div> See brief description of the methods BOXMULLER , BOXMULLER2 , and ICDF in Table "Values of <method> in method parameter"
stream	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr	Fortran: Descriptor of the stream state structure C: Pointer to the stream state structure

Name	Type	Description
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Number of random values to be generated
<i>a</i>	FORTRAN 77: REAL for vsrnggaussian DOUBLE PRECISION for vdrnggaussian Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnggaussian REAL(KIND=8), INTENT(IN) for vdrnggaussian C: const float for vsRngGaussian const double for vdRngGaussian	Mean value a .
<i>sigma</i>	FORTRAN 77: REAL for vsrnggaussian DOUBLE PRECISION for vdrnggaussian Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnggaussian REAL(KIND=8), INTENT(IN) for vdrnggaussian C: const float for vsRngGaussian const double for vdRngGaussian	Standard deviation σ .

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: REAL for vsrnggaussian DOUBLE PRECISION for vdrnggaussian Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrnggaussian REAL(KIND=8), INTENT(OUT) for vdrnggaussian C: float* for vsRngGaussian	Vector of <i>n</i> normally distributed random numbers

Name	Type	Description
	double* for vdRngGaussian	

Description

The vdRngGaussian function generates random numbers with normal (Gaussian) distribution with mean value a and standard deviation σ , where

$$a, \sigma \in \mathbb{R} ; \sigma > 0.$$

The probability density function is given by:

$$f_{a, \sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - a)^2}{2\sigma^2}\right), \quad -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a, \sigma}(x) = \int_{-\infty}^x \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y - a)^2}{2\sigma^2}\right) dy, \quad -\infty < x < +\infty.$$

The cumulative distribution function $F_{a, \sigma}(x)$ can be expressed in terms of standard normal distribution $\Phi(x)$ as

$$F_{a, \sigma}(x) = \Phi((x - a)/\sigma)$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngGaussianMV

Generates random numbers from multivariate normal distribution.

Syntax

Fortran:

```
status = vsrnggaussianmv( method, stream, n, r, dimen, mstorage, a, t )
status = vdrgngaussianmv( method, stream, n, r, dimen, mstorage, a, t )
```

C:

```
status = vsRngGaussianMV( method, stream, n, r, dimen, mstorage, a, t );
status = vdRngGaussianMV( method, stream, n, r, dimen, mstorage, a, t );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Generation method. The specific values are as follows: <div style="background-color: #f0f0f0; padding: 2px;">VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER</div> <div style="background-color: #f0f0f0; padding: 2px;">VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2</div> <div style="background-color: #f0f0f0; padding: 2px;">VSL_RNG_METHOD_GAUSSIANMV_ICDF</div>
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	See brief description of the methods BOXMULLER, BOXMULLER2, and ICDF in Table "Values of <method> in method parameter" Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Number of d -dimensional vectors to be generated
<i>dimen</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Dimension d ($d \geq 1$) of output random vectors
<i>mstorage</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Fortran: Matrix storage scheme for upper triangular matrix T^T . The routine supports three matrix storage schemes:

Name	Type	Description
		<ul style="list-style-type: none"> • VSL_MATRIX_STORAGE_FULL— all $d \times d$ elements of the matrix T^T are passed, however, only the upper triangle part is actually used in the routine. • VSL_MATRIX_STORAGE_PACKED— upper triangle elements of T^T are packed by rows into a one-dimensional array. • VSL_MATRIX_STORAGE_DIAGONAL— only diagonal elements of T^T are passed. <p>C: Matrix storage scheme for lower triangular matrix T. The routine supports three matrix storage schemes:</p> <ul style="list-style-type: none"> • VSL_MATRIX_STORAGE_FULL— all $d \times d$ elements of the matrix T are passed, however, only the lower triangle part is actually used in the routine. • VSL_MATRIX_STORAGE_PACKED— lower triangle elements of T are packed by rows into a one-dimensional array. • VSL_MATRIX_STORAGE_DIAGONAL— only diagonal elements of T are passed.
<i>a</i>	<pre>FORTRAN 77: REAL for vsrnggaussianmv DOUBLE PRECISION for vdrnggaussianmv Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnggaussianmv REAL(KIND=8), INTENT(IN) for vdrnggaussianmv C: const float* for vsRngGaussianMV const double* for vdRngGaussianMV</pre>	Mean vector <i>a</i> of dimension <i>d</i>
<i>t</i>	<pre>FORTRAN 77: REAL for vsrnggaussianmv DOUBLE PRECISION for vdrnggaussianmv Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnggaussianmv REAL(KIND=8), INTENT(IN) for vdrnggaussianmv C: const float* for vsRngGaussianMV</pre>	<p>Fortran: Elements of the upper triangular matrix passed according to the matrix T^T storage scheme <i>mstorage</i>.</p> <p>C: Elements of the lower triangular matrix passed according to the matrix T storage scheme <i>mstorage</i>.</p>

Name	Type	Description
	const double* for vdRngGaussianMV	

Output Parameters

Name	Type	Description
r	FORTRAN 77: REAL for vsrnggaussianmv DOUBLE PRECISION for vdrnggaussianmv Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrnggaussianmv REAL(KIND=8), INTENT(OUT) for vdrnggaussianmv C: float* for vsRngGaussianMV double* for vdRngGaussianMV	Array of n random vectors of dimension $dimen$

Description

The `vRngGaussianMV` function generates random numbers with d -variate normal (Gaussian) distribution with mean value a and variance-covariance matrix C , where $a \in R^d$; C is a $d \times d$ symmetric positive-definite matrix.

The probability density function is given by:

$$f_{a, C}(x) = \frac{1}{\sqrt{\det(2\pi C)}} \exp(-1/2(x - a)^T C^{-1} (x - a)) ,$$

where $x \in R^d$.

Matrix C can be represented as $C = TT^T$, where T is a lower triangular matrix - Cholesky factor of C .

Instead of variance-covariance matrix C the generation routines require Cholesky factor of C in input. To compute Cholesky factor of matrix C , the user may call Intel MKL LAPACK routines for matrix factorization: `?potrf` or `?pptrf` for `v?RngGaussianMV/v?rnggaussianmv` routines (`? means either s or d for single and double precision respectively). See Application Notes for more details.`

Application Notes

Since matrices are stored in Fortran by columns, while in C they are stored by rows, the usage of Intel MKL factorization routines (assuming Fortran matrices storage) in combination with multivariate normal RNG (assuming C matrix storage) is slightly different in C and Fortran. The following tables help in using these routines in C and Fortran. For further information please refer to the appropriate VSL example file.

Using Cholesky Factorization Routines in Fortran

Matrix Storage Scheme	Variance-Covariance Matrix Argument	Factorization Routine	UPLO Parameter in Factorization Routine	Result of Factorization as Input Argument for RNG
VSL_MATRIX_STORAGE_FULL	C in Fortran two-dimensional array	spotrf for vsrnggaussianmv dpotrf for vdrnggaussianmv	'U'	Upper triangle of T^T . Lower triangle is not used.
VSL_MATRIX_STORAGE_PACKED	Lower triangle of C packed by columns into one-dimensional array	spptrf for vsrnggaussianmv dpptrf for vdrnggaussianmv	'L'	Upper triangle of T^T packed by rows into a one-dimensional array.

Using Cholesky Factorization Routines in C

Matrix Storage Scheme	Variance-Covariance Matrix Argument	Factorization Routine	UPLO Parameter in Factorization Routine	Result of Factorization as Input Argument for RNG
VSL_MATRIX_STORAGE_FULL	C in C two-dimensional array	spotrf for vsRngGaussianMV dpotrf for vdRngGaussianMV	'U'	Lower triangle of T . Upper triangle is not used.
VSL_MATRIX_STORAGE_PACKED	Lower triangle of C packed by columns into one-dimensional array	spptrf for vsRngGaussianMV dpptrf for vdRngGaussianMV	'L'	Lower triangle of T packed by columns into a one-dimensional array.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngExponential

Generates exponentially distributed random numbers.

Syntax

Fortran:

```
status = vsrngexponential( method, stream, n, r, a, beta )
status = vdrngexponential( method, stream, n, r, a, beta )
```

C:

```
status = vsRngExponential( method, stream, n, r, a, beta );
status = vdRngExponential( method, stream, n, r, a, beta );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
method	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_EXPONENTIAL_ICDF VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE
stream	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr	Inverse cumulative distribution function method Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Number of random values to be generated
a	FORTRAN 77: REAL for vsrngexponential	Displacement a

Name	Type	Description
	<p>DOUBLE PRECISION for vdrngexponential</p> <p>Fortran 90: REAL(KIND=4), INTENT(IN) for vsrngexponential</p> <p>REAL(KIND=8), INTENT(IN) for vdrngexponential</p> <p>C: const float for vsRngExponential</p> <p>C: const double for vdRngExponential</p>	
beta	<p>FORTRAN 77: REAL for vsrngexponential</p> <p>DOUBLE PRECISION for vdrngexponential</p> <p>Fortran 90: REAL(KIND=4), INTENT(IN) for vsrngexponential</p> <p>REAL(KIND=8), INTENT(IN) for vdrngexponential</p> <p>C: const float for vsRngExponential</p> <p>const double for vdRngExponential</p>	Scalefactor β .

Output Parameters

Name	Type	Description
r	<p>FORTRAN 77: REAL for vsrngexponential</p> <p>DOUBLE PRECISION for vdrngexponential</p> <p>Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrngexponential</p> <p>REAL(KIND=8), INTENT(OUT) for vdrngexponential</p> <p>C: float* for vsRngExponential</p> <p>double* for vdRngExponential</p>	Vector of n exponentially distributed random numbers

Description

The `vRngExponential` function generates random numbers with exponential distribution that has displacement a and scalefactor β , where $a, \beta \in \mathbb{R} ; \beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{1}{\beta} \exp(-(x - a)/\beta), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp(-(x - a)/\beta), & x \geq a \\ 0, & x < a \end{cases}, -\infty < x < +\infty.$$

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<code>stream</code> is a NULL pointer.
<code>VSL RNG ERROR BAD STREAM</code>	<code>stream</code> is not a valid random stream.
<code>VSL RNG ERROR BAD UPDATE</code>	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
<code>VSL RNG ERROR NO NUMBERS</code>	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
<code>VSL RNG ERROR QRNG PERIOD ELAPSED</code>	Period of the generator has been exceeded.
<code>VSL RNG ERROR NONDETERM_NRETRIES_EXCEEDED</code>	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

`vRngLaplace`

Generates random numbers with Laplace distribution.

Syntax

Fortran:

```
status = vsrnglaplace( method, stream, n, r, a, beta )
status = vdrglaplace( method, stream, n, r, a, beta )
```

C:

```
status = vsRngLaplace( method, stream, n, r, a, beta );
status = vdRngLaplace( method, stream, n, r, a, beta );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT</pre>	Generation method. The specific values are as follows: VSL_RNG_METHOD_LAPLACE_ICDF
		Inverse cumulative distribution function method
<i>stream</i>	<pre>FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr</pre>	Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT</pre>	Number of random values to be generated
<i>a</i>	<pre>FORTRAN 77: REAL for vsrnglaplace DOUBLE PRECISION for vdrnglaplace Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnglaplace REAL(KIND=8), INTENT(IN) for vdrnglaplace C: const float for vsRngLaplace const double for vdRngLaplace</pre>	Mean value <i>a</i>
<i>beta</i>	<pre>FORTRAN 77: REAL for vsrnglaplace DOUBLE PRECISION for vdrnglaplace Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnglaplace REAL(KIND=8), INTENT(IN) for vdrnglaplace</pre>	Scalefactor β .

Name	Type	Description
	C: const float for vsRngLaplace	
	const double for vdRngLaplace	

Output Parameters

Name	Type	Description
r	FORTRAN 77: REAL for vsrnglaplace DOUBLE PRECISION for vdrnglaplace Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrnglaplace REAL(KIND=8), INTENT(OUT) for vdrnglaplace C: float* for vsRngLaplace double* for vdRngLaplace	Vector of n Laplace distributed random numbers

Description

The `vRngLaplace` function generates random numbers with Laplace distribution with mean value (or average) α and scalefactor β , where $\alpha, \beta \in \mathbb{R}$; $\beta > 0$. The scalefactor value determines the standard deviation as

.....

The probability density function is given by:

$$f_{\alpha, \beta}(x) = \frac{1}{\sqrt{2\beta}} \exp\left(-\frac{|x - \alpha|}{\beta}\right), -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{\alpha, \beta}(x) = \begin{cases} \frac{1}{2} \exp\left(-\frac{|x - \alpha|}{\beta}\right), & x \geq \alpha \\ 1 - \frac{1}{2} \exp\left(-\frac{|x - \alpha|}{\beta}\right), & x < \alpha \end{cases}, -\infty < x < +\infty.$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngWeibull

Generates Weibull distributed random numbers.

Syntax

Fortran:

```
status = vsrngweibull( method, stream, n, r, alpha, a, beta )
status = vdrngweibull( method, stream, n, r, alpha, a, beta )
```

C:

```
status = vsRngWeibull( method, stream, n, r, alpha, a, beta );
status = vdRngWeibull( method, stream, n, r, alpha, a, beta );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
method	FORTRAN 77 : INTEGER Fortran 90 : INTEGER, INTENT(IN) C : const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_WEIBULL_ICDF VSL_RNG_METHOD_WEIBULL_ICDF_ACCURATE
stream	FORTRAN 77 : INTEGER*4 stream(2)	Inverse cumulative distribution function method Fortran : Descriptor of the stream state structure. C : Pointer to the stream state structure

Name	Type	Description
	Fortran 90: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Number of random values to be generated
<i>alpha</i>	FORTRAN 77: REAL for vsrngweibull DOUBLE PRECISION for vdrngweibull Fortran 90: REAL(KIND=4), INTENT (IN) for vsrngweibull REAL(KIND=8), INTENT (IN) for vdrngweibull C: const float for vsRngWeibull const double for vdRngWeibull	Shape α .
<i>a</i>	FORTRAN 77: REAL for vsrngweibull DOUBLE PRECISION for vdrngweibull Fortran 90: REAL(KIND=4), INTENT (IN) for vsrngweibull REAL(KIND=8), INTENT (IN) for vdrngweibull C: const float for vsRngWeibull const double for vdRngWeibull	Displacement a
<i>beta</i>	FORTRAN 77: REAL for vsrngweibull DOUBLE PRECISION for vdrngweibull Fortran 90: REAL(KIND=4), INTENT (IN) for vsrngweibull REAL(KIND=8), INTENT (IN) for vdrngweibull	Scalefactor β .

Name	Type	Description
	C: const float for vsRngWeibull	
	const double for vdRngWeibull	

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: REAL for vsrngweibull DOUBLE PRECISION for vdrngweibull Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrngweibull REAL(KIND=8), INTENT(OUT) for vdrngweibull C: float* for vsRngWeibull double* for vdRngWeibull	Vector of <i>n</i> Weibull distributed random numbers

Description

The `vRngWeibull` function generates Weibull distributed random numbers with displacement a , scalefactor β , and shape α , where $\alpha, \beta, a \in R ; \alpha > 0, \beta > 0$.

The probability density function is given by:

$$f_{\alpha, \beta, a}(x) = \begin{cases} \frac{\alpha}{\beta^\alpha} (x - a)^{\alpha-1} \exp\left(-\left(\frac{x-a}{\beta}\right)^\alpha\right), & x \geq a \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{\alpha, \beta, a}(x) = \begin{cases} 1 - \exp\left(-\left(\frac{x-a}{\beta}\right)^\alpha\right), & x \geq a, -\infty < x < +\infty. \\ 0, & x < a \end{cases}$$

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`

Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngCauchy

Generates Cauchy distributed random values.

Syntax

Fortran:

```
status = vsrngcauchy( method, stream, n, r, a, beta )
status = vdrngcauchy( method, stream, n, r, a, beta )
```

C:

```
status = vsRngCauchy( method, stream, n, r, a, beta );
status = vdRngCauchy( method, stream, n, r, a, beta );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
method	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_CAUCHY_ICDF Inverse cumulative distribution function method
stream	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr	Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure

Name	Type	Description
<i>n</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT</pre>	Number of random values to be generated
<i>a</i>	<pre>FORTRAN 77: REAL for vsrngcauchy DOUBLE PRECISION for vdrngcauchy Fortran 90: REAL(KIND=4), INTENT(IN) for vsrngcauchy REAL(KIND=8), INTENT(IN) for vdrngcauchy C: const float for vsRngCauchy const double for vdRngCauchy</pre>	Displacement <i>a</i> .
<i>beta</i>	<pre>FORTRAN 77: REAL for vsrngcauchy DOUBLE PRECISION for vdrngcauchy Fortran 90: REAL(KIND=4), INTENT(IN) for vsrngcauchy REAL(KIND=8), INTENT(IN) for vdrngcauchy C: const float for vsRngCauchy const double for vdRngCauchy</pre>	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	<pre>FORTRAN 77: REAL for vsrngcauchy DOUBLE PRECISION for vdrngcauchy Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrngcauchy REAL(KIND=8), INTENT(OUT) for vdrngcauchy C: float* for vsRngCauchy double* for vdRngCauchy</pre>	Vector of <i>n</i> Cauchy distributed random numbers

Description

The function generates Cauchy distributed random numbers with displacement a and scalefactor β , where $a, \beta \in R$; $\beta > 0$.

The probability density function is given by:

$$f_{a,\beta}(x) = \frac{1}{\pi\beta \left(1 + \left(\frac{x-a}{\beta}\right)^2\right)}, -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \frac{1}{2} + \frac{1}{\pi} \arctan\left(\frac{x-a}{\beta}\right), -\infty < x < +\infty.$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngRayleigh

Generates Rayleigh distributed random values.

Syntax

Fortran:

```
status = vsrngrayleigh( method, stream, n, r, a, beta )
status = vdrngrayleigh( method, stream, n, r, a, beta )
```

C:

```
status = vsRngRayleigh( method, stream, n, r, a, beta );
status = vdRngRayleigh( method, stream, n, r, a, beta );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT</pre>	Generation method. The specific values are as follows: <pre>VSL_RNG_METHOD_RAYLEIGH_ICDF VSL_RNG_METHOD_RAYLEIGH_ICDF_ACCURATE</pre>
<i>stream</i>	<pre>FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr</pre>	Inverse cumulative distribution function method <pre>Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure</pre>
<i>n</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT</pre>	Number of random values to be generated
<i>a</i>	<pre>FORTRAN 77: REAL for vsrngrayleigh DOUBLE PRECISION for vdrngrayleigh Fortran 90: REAL(KIND=4), INTENT(IN) for vsrngrayleigh REAL(KIND=8), INTENT(IN) for vdrngrayleigh C: const float for vsRngRayleigh const double for vdRngRayleigh</pre>	Displacement <i>a</i>
<i>beta</i>	<pre>FORTRAN 77: REAL for vsrngrayleigh DOUBLE PRECISION for vdrngrayleigh Fortran 90: REAL(KIND=4), INTENT(IN) for vsrngrayleigh REAL(KIND=8), INTENT(IN) for vdrngrayleigh</pre>	Scalefactor β .

Name	Type	Description
	C: const float for vsRngRayleigh	
	const double for vdRngRayleigh	

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: REAL for vsrngrayleigh DOUBLE PRECISION for vdrngrayleigh Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrngrayleigh REAL(KIND=8), INTENT(OUT) for vdrngrayleigh C: float* for vsRngRayleigh double* for vdRngRayleigh	Vector of <i>n</i> Rayleigh distributed random numbers

Description

The `vRngRayleigh` function generates Rayleigh distributed random numbers with displacement a and scalefactor β , where $a, \beta \in R ; \beta > 0$.

The Rayleigh distribution is a special case of the [Weibull](#) distribution, where the shape parameter $\alpha = 2$.

The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{2(x - a)}{\beta^2} \exp\left(-\frac{(x - a)^2}{\beta^2}\right), & x \geq a, -\infty < x < +\infty. \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp\left(-\frac{(x - a)^2}{\beta^2}\right), & x \geq a, -\infty < x < +\infty. \\ 0, & x < a \end{cases}$$

Return Values

`VSL_ERROR_OK, VSL_STATUS_OK`

Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngLognormal

Generates lognormally distributed random numbers.

Syntax

Fortran:

```
status = vsrnglognormal( method, stream, n, r, a, sigma, b, beta )
status = vdrnglognormal( method, stream, n, r, a, sigma, b, beta )
```

C:

```
status = vsRngLognormal( method, stream, n, r, a, sigma, b, beta );
status = vdRngLognormal( method, stream, n, r, a, sigma, b, beta );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2 VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2_ACCURATE Box Muller 2 based method VSL_RNG_METHOD_LOGNORMAL_ICDF VSL_RNG_METHOD_LOGNORMAL_ICDF_ACCURATE Inverse cumulative distribution function based method
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2)	Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure

Name	Type	Description
	<pre>Fortran 90: TYPE (VSL_STREAM_STATE), INTENT (IN)</pre> <p>C: VSLStreamStatePtr</p>	
<i>n</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN)</pre> <p>C: const MKL_INT</p>	Number of random values to be generated
<i>a</i>	<pre>FORTRAN 77: REAL for vsrnglognormal DOUBLE PRECISION for vdrnglognormal Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnglognormal REAL(KIND=8), INTENT(IN) for vdrnglognormal C: const float for vsRngLognormal const double for vdRngLognormal</pre>	Average <i>a</i> of the subject normal distribution
<i>sigma</i>	<pre>FORTRAN 77: REAL for vsrnglognormal DOUBLE PRECISION for vdrnglognormal Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnglognormal REAL(KIND=8), INTENT(IN) for vdrnglognormal C: const float for vsRngLognormal const double for vdRngLognormal</pre>	Standard deviation σ of the subject normal distribution
<i>b</i>	<pre>FORTRAN 77: REAL for vsrnglognormal DOUBLE PRECISION for vdrnglognormal Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnglognormal</pre>	Displacement <i>b</i>

Name	Type	Description
	REAL(KIND=8), INTENT(IN) for vdrnglognormal C: const float for vsRngLognormal const double for vdRngLognormal	
beta	FORTRAN 77: REAL for vsrnglognormal DOUBLE PRECISION for vdrnglognormal Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnglognormal REAL(KIND=8), INTENT(IN) for vdrnglognormal C: const float for vsRngLognormal const double for vdRngLognormal	Scalefactor β .

Output Parameters

Name	Type	Description
r	FORTRAN 77: REAL for vsrnglognormal DOUBLE PRECISION for vdrnglognormal Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrnglognormal REAL(KIND=8), INTENT(OUT) for vdrnglognormal C: float* for vsRngLognormal double* for vdRngLognormal	Vector of n lognormally distributed random numbers

Description

The `vRngLognormal` function generates lognormally distributed random numbers with average of distribution a and standard deviation σ of subject normal distribution, displacement b , and scalefactor β , where $a, \sigma, b, \beta \in \mathbb{R}$; $\sigma > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{a,\sigma,b,\beta}(x) = \begin{cases} \frac{1}{\sigma(x - b)\sqrt{2\pi}} \exp\left(-\frac{[\ln((x - b)/\beta) - a]^2}{2\sigma^2}\right), & x > b \\ 0, & x \leq b \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\sigma,b,\beta}(x) = \begin{cases} \Phi(\ln((x - b)/\beta) - a)/\sigma, & x > b \\ 0, & x \leq b \end{cases}$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngGumbel

Generates Gumbel distributed random values.

Syntax

Fortran:

```
status = vsrnggumbel( method, stream, n, r, a, beta )
status = vdrgngumbel( method, stream, n, r, a, beta )
```

C:

```
status = vsRngGumbel( method, stream, n, r, a, beta );
status = vdRngGumbel( method, stream, n, r, a, beta );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90

- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_GUMBEL_ICDF
		Inverse cumulative distribution function method
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	Fortran: Descriptor of the stream state structure C: Pointer to the stream state structure
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Number of random values to be generated
<i>a</i>	FORTRAN 77: REAL for vsrnggumbel DOUBLE PRECISION for vdrnggumbel Fortran 90: REAL(KIND=4), INTENT (IN) for vsrnggumbel REAL(KIND=8), INTENT (IN) for vdrnggumbel C: const float for vsRngGumbel const double for vdRngGumbel	Displacement a .
<i>beta</i>	FORTRAN 77: REAL for vsrnggumbel DOUBLE PRECISION for vdrnggumbel Fortran 90: REAL(KIND=4), INTENT (IN) for vsrnggumbel REAL(KIND=8), INTENT (IN) for vdrnggumbel C: const float for vsRngGumbel const double for vdRngGumbel	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: REAL for vsrnggumbel DOUBLE PRECISION for vdrnggumbel Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrnggumbel REAL(KIND=8), INTENT(OUT) for vdrnggumbel C: float* for vsRngGumbel double* for vdRngGumbel	Vector of <i>n</i> random numbers with Gumbel distribution

Description

The `vRngGumbel` function generates Gumbel distributed random numbers with displacement α and scalefactor β , where $\alpha, \beta \in R ; \beta > 0$.

The probability density function is given by:

$$f_{\alpha, \beta}(x) = \frac{1}{\beta} \exp\left(\frac{x - \alpha}{\beta}\right) \exp(-\exp((x - \alpha) / \beta)), -\infty < x < +\infty.$$

The cumulative distribution function is as follows:

$$F_{\alpha, \beta}(x) = 1 - \exp(-\exp((x - \alpha) / \beta)), -\infty < x < +\infty$$

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<i>stream</i> is a NULL pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>stream</i> is not a valid random stream.
<code>VSL_RNG_ERROR_BAD_UPDATE</code>	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
<code>VSL_RNG_ERROR_NO_NUMBERS</code>	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
<code>VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED</code>	Period of the generator has been exceeded.

`VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED` Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngGamma

Generates gamma distributed random values.

Syntax

Fortran:

```
status = vsrnggamma( method, stream, n, r, alpha, a, beta )
status = vdrnggamma( method, stream, n, r, alpha, a, beta )
```

C:

```
status = vsRngGamma( method, stream, n, r, alpha, a, beta );
status = vdRngGamma( method, stream, n, r, alpha, a, beta );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Generation method. The specific values are as follows: <code>VSL_RNG_METHOD_GAMMA_GNORM</code> <code>VSL_RNG_METHOD_GAMMA_GNORM_ACCURATE</code> Acceptance/rejection method using random numbers with Gaussian distribution. See brief description of the method GNORM in Table "Values of <method> in method parameter"
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr	Fortran: Descriptor of the stream state structure C: Pointer to the stream state structure
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Number of random values to be generated
<i>alpha</i>	FORTRAN 77: REAL for vsrnggamma DOUBLE PRECISION for vdrnggamma	Shape α .

Name	Type	Description
	<pre> Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnggamma REAL(KIND=8), INTENT(IN) for vdrnggamma C: const float for vsRngGamma const double for vdRngGamma </pre>	
a	<pre> FORTRAN 77: REAL for Displacement a. vsrnggamma DOUBLE PRECISION for vdrnggamma Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnggamma REAL(KIND=8), INTENT(IN) for vdrnggamma C: const float for vsRngGamma const double for vdRngGamma </pre>	
beta	<pre> FORTRAN 77: REAL for Scalefactor β. vsrnggamma DOUBLE PRECISION for vdrnggamma Fortran 90: REAL(KIND=4), INTENT(IN) for vsrnggamma REAL(KIND=8), INTENT(IN) for vdrnggamma C: const float for vsRngGamma const double for vdRngGamma </pre>	

Output Parameters

Name	Type	Description
r	<pre> FORTRAN 77: REAL for Vector of n random numbers with gamma distribution vsrnggamma DOUBLE PRECISION for vdrnggamma Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrnggamma REAL(KIND=8), INTENT(OUT) for vdrnggamma </pre>	

Name	Type	Description
	C: float* for vsRngGamma double* for vdRngGamma	

Description

The `vRngGamma` function generates random numbers with gamma distribution that has shape parameter α , displacement a , and scale parameter β , where α, β , and $a \in \mathbb{R}$; $\alpha > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{\alpha,a,\beta}(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^\alpha} (x - a)^{\alpha-1} e^{-(x-a)/\beta}, & x \geq a \\ 0, & x < a \end{cases}, \quad -\infty < x < +\infty$$

where $\Gamma(\alpha)$ is the complete gamma function.

The cumulative distribution function is as follows:

$$F_{\alpha,a,\beta}(x) = \begin{cases} \int_a^x \frac{1}{\Gamma(\alpha)\beta^\alpha} (y - a)^{\alpha-1} e^{-(y-a)/\beta} dy, & x \geq a \\ 0, & x < a \end{cases}, \quad -\infty < x < +\infty$$

Return Values

<code>VSL_ERROR_OK</code> , <code>VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<i>stream</i> is a NULL pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>stream</i> is not a valid random stream.
<code>VSL_RNG_ERROR_BAD_UPDATE</code>	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
<code>VSL_RNG_ERROR_NO_NUMBERS</code>	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
<code>VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED</code>	Period of the generator has been exceeded.
<code>VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED</code>	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngBeta

Generates beta distributed random values.

Syntax

Fortran:

```
status = vsrngbeta( method, stream, n, r, p, q, a, beta )
```

```
status = vdrngbeta( method, stream, n, r, p, q, a, beta )
```

C:

```
status = vsRngBeta( method, stream, n, r, p, q, a, beta );
status = vdRngBeta( method, stream, n, r, p, q, a, beta );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Generation method. The specific values are as follows: <div style="background-color: #f0f0f0; padding: 5px;">VSL_RNG_METHOD_BETA_CJA</div> <div style="background-color: #f0f0f0; padding: 5px;">VSL_RNG_METHOD_BETA_CJA_ACCURATE</div>
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr	Fortran: Descriptor of the stream state structure C: Pointer to the stream state structure
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Number of random values to be generated
<i>p</i>	FORTRAN 77: REAL for vsrngbeta DOUBLE PRECISION for vdrngbeta Fortran 90: REAL(KIND=4), INTENT(IN) for vsrngbeta REAL(KIND=8), INTENT(IN) for vdrngbeta C: const float for vsRngBeta const double for vdRngBeta	Shape <i>p</i>
<i>q</i>	FORTRAN 77: REAL for vsrngbeta DOUBLE PRECISION for vdrngbeta	Shape <i>q</i>

Name	Type	Description
	<pre>Fortran 90: REAL(KIND=4), INTENT(IN) for vsrngbeta</pre> <pre>REAL(KIND=8), INTENT(IN) for vdrngbeta</pre> <pre>C: const float for vsRngBeta</pre> <pre>const double for vdRngBeta</pre>	
<i>a</i>	<pre>FORTRAN 77: REAL for vsrngbeta</pre> <pre>DOUBLE PRECISION for vdrngbeta</pre> <pre>Fortran 90: REAL(KIND=4), INTENT(IN) for vsrngbeta</pre> <pre>REAL(KIND=8), INTENT(IN) for vdrngbeta</pre> <pre>C: const float for vsRngBeta</pre> <pre>const double for vdRngBeta</pre>	Displacement <i>a</i> .
<i>beta</i>	<pre>FORTRAN 77: REAL for vsrngbeta</pre> <pre>DOUBLE PRECISION for vdrngbeta</pre> <pre>Fortran 90: REAL(KIND=4), INTENT(IN) for vsrngbeta</pre> <pre>REAL(KIND=8), INTENT(IN) for vdrngbeta</pre> <pre>C: const float for vsRngBeta</pre> <pre>const double for vdRngBeta</pre>	Scalefactor β .

Output Parameters

Name	Type	Description
<i>r</i>	<pre>FORTRAN 77: REAL for vsrngbeta</pre> <pre>DOUBLE PRECISION for vdrngbeta</pre> <pre>Fortran 90: REAL(KIND=4), INTENT(OUT) for vsrngbeta</pre> <pre>REAL(KIND=8), INTENT(OUT) for vdrngbeta</pre> <pre>C: float* for vsRngBeta</pre> <pre>double* for vdRngBeta</pre>	Vector of <i>n</i> random numbers with beta distribution

Description

The `vRngBeta` function generates random numbers with beta distribution that has shape parameters p and q , displacement a , and scale parameter β , where p, q, a , and $\beta \in \mathbb{R}$; $p > 0, q > 0, \beta > 0$.

The probability density function is given by:

$$f_{p,q,a,\beta}(x) = \begin{cases} \frac{1}{B(p, q)\beta^{p+q-1}} (x - a)^{p-1}(\beta + a - x)^{q-1}, & a \leq x < a + \beta \\ 0, & x < a, x \geq a + \beta \end{cases}, \quad -\infty < x < \infty,$$

where $B(p, q)$ is the complete beta function.

The cumulative distribution function is as follows:

$$F_{p,q,a,\beta}(x) = \begin{cases} 0, & x < a \\ \int_a^x \frac{1}{B(p, q)\beta^{p+q-1}} (y - a)^{p-1}(\beta + a - y)^{q-1} dy, & a \leq x < a + \beta, -\infty < x < \infty. \\ 1, & x \geq a + \beta \end{cases}$$

Return Values

<code>VSL_ERROR_OK, VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<i>stream</i> is a NULL pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>stream</i> is not a valid random stream.
<code>VSL_RNG_ERROR_BAD_UPDATE</code>	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
<code>VSL_RNG_ERROR_NO_NUMBERS</code>	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
<code>VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED</code>	Period of the generator has been exceeded.
<code>VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED</code>	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

Discrete Distributions

This section describes routines for generating random numbers with discrete distribution.

`vRngUniform`

Generates random numbers uniformly distributed over the interval $[a, b]$.

Syntax

Fortran:

```
status = virnguniform( method, stream, n, r, a, b )
```

C:

```
status = viRngUniform( method, stream, n, r, a, b );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
method	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Generation method; the specific value is as follows: VSL_RNG_METHOD_UNIFORM_STD Standard method. Currently there is only one method for this distribution generator.
stream	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr	Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Number of random values to be generated
a	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(IN) C: const int	Left interval bound a
b	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(IN) C: const int	Right interval bound b

Output Parameters

Name	Type	Description
r	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(OUT) C: int*	Vector of n random numbers uniformly distributed over the interval [a, b)

Description

The `vRngUniform` function generates random numbers uniformly distributed over the interval $[a, b]$, where a, b are the left and right bounds of the interval respectively, and $a, b \in \mathbb{Z}; a < b$.

The probability distribution is given by:

$$P(X = k) = \frac{1}{b - a}, k \in \{a, a + 1, \dots, b - 1\}.$$

The cumulative distribution function is as follows:

$$F_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{\lfloor x - a + 1 \rfloor}{b - a}, & a \leq x < b, x \in \mathbb{R} \\ 1, & x \geq b \end{cases}$$

Return Values

<code>VSL_ERROR_OK</code> , <code>VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<code>stream</code> is a NULL pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<code>stream</code> is not a valid random stream.
<code>VSL_RNG_ERROR_BAD_UPDATE</code>	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
<code>VSL_RNG_ERROR_NO_NUMBERS</code>	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
<code>VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED</code>	Period of the generator has been exceeded.
<code>VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED</code>	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

`vRngUniformBits`

Generates bits of underlying BRNG integer recurrence.

Syntax

Fortran:

```
status = virnguniformbits( method, stream, n, r )
```

C:

```
status = viRngUniformBits( method, stream, n, r );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Generation method; the specific value is VSL_RNG_METHOD_UNIFORMBITS_STD
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Number of random values to be generated

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER (KIND=4), INTENT (OUT) C: unsigned int*	Fortran: Vector of <i>n</i> random integer numbers. If the <i>stream</i> was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of <i>r</i> respectively. The number of bytes occupied by each integer is contained in the field <i>wordsize</i> of the structure VSL_BRNG_PROPERTIES . The total number of bits that are actually used to store the value are contained in the field <i>nbits</i> of the same structure. See Advanced Service Routines for a more detailed discussion of VSLBRngProperties . C: Vector of <i>n</i> random integer numbers. If the <i>stream</i> was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of <i>r</i> respectively. The number of bytes occupied by each integer is contained in the field <i>WordSize</i> of the structure VSLBRngProperties . The total number of bits that are actually used to store the value are contained in the field <i>NBits</i> of the same structure. See Advanced Service Routines for a more detailed discussion of VSLBRngProperties .

Description

The `vRngUniformBits` function generates integer random values with uniform bit distribution. The generators of uniformly distributed numbers can be represented as recurrence relations over integer values in modular arithmetic. Apparently, each integer can be treated as a vector of several bits. In a truly random generator, these bits are random, while in pseudorandom generators this randomness can be violated. For example, a well known drawback of linear congruential generators is that lower bits are less random than higher bits (for example, see [Knuth81]). For this reason, care should be taken when using this function. Typically, in a 32-bit LCG only 24 higher bits of an integer value can be considered random. See [VSL Notes](#) for details.

Return Values

<code>VSL_ERROR_OK</code> , <code>VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<code>stream</code> is a NULL pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<code>stream</code> is not a valid random stream.
<code>VSL_RNG_ERROR_BAD_UPDATE</code>	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
<code>VSL_RNG_ERROR_NO_NUMBERS</code>	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
<code>VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED</code>	Period of the generator has been exceeded.
<code>VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED</code>	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

`vRngUniformBits32`

Generates uniformly distributed bits in 32-bit chunks.

Syntax

Fortran:

```
status = virnguniformbits32( method, stream, n, r )
```

C:

```
status = viRngUniformBits32( method, stream, n, r );
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>method</code>	<code>FORTRAN 77:</code> INTEGER <code>Fortran 90:</code> INTEGER, INTENT (IN)	Generation method; the specific value is <code>VSL_RNG_METHOD_UNIFORMBITS32_STD</code>

Name	Type	Description
	C: const MKL_INT	
stream	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT (IN)	Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
	C: VSLStreamStatePtr	
n	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Number of random values to be generated

Output Parameters

Name	Type	Description
r	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER (KIND=4), INTENT (OUT) C: unsigned int*	Fortran: Vector of n 32-bit random integer numbers with uniform bit distribution. C: Vector of n 32-bit random integer numbers with uniform bit distribution.

Description

The `vRngUniformBits32` function generates uniformly distributed bits in 32-bit chunks. Unlike `vRngUniformBits`, which provides the output of underlying integer recurrence and does not guarantee uniform distribution across bits, `vRngUniformBits32` is designed to ensure each bit in the 32-bit chunk is uniformly distributed. See [VSL Notes](#) for details.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<code>stream</code> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<code>stream</code> is not a valid random stream.
VSL_RNG_ERROR_BRNG_NOT_SUPPORTED	BRNG is not supported by the function.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngUniformBits64

Generates uniformly distributed bits in 64-bit chunks.

Syntax

Fortran:

```
status = virnguniformbits64( method, stream, n, r )
```

C:

```
status = viRngUniformBits64( method, stream, n, r );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Generation method; the specific value is VSL_RNG_METHOD_UNIFORMBITS64_STD
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr	Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Number of random values to be generated

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: INTEGER*8 Fortran 90: INTEGER (KIND=8), INTENT(OUT) C: unsigned MKL_INT64*	Fortran: Vector of <i>n</i> 64-bit random integer numbers with uniform bit distribution. C: Vector of <i>n</i> 64-bit random integer numbers with uniform bit distribution.

Description

The vRngUniformBits64 function generates uniformly distributed bits in 64-bit chunks. Unlike vRngUniformBits, which provides the output of underlying integer recurrence and does not guarantee uniform distribution across bits, vRngUniformBits64 is designed to ensure each bit in the 64-bit chunk is uniformly distributed. See [VSL Notes](#) for details.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BRNG_NOT_SUPPORTED	BRNG is not supported by the function.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngBernoulli

Generates Bernoulli distributed random values.

Syntax

Fortran:

```
status = virngbernoulli( method, stream, n, r, p )
```

C:

```
status = viRngBernoulli( method, stream, n, r, p );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Generation method. The specific value is as follows: VSL_RNG_METHOD_BERNOULLI_ICDF Inverse cumulative distribution function method.
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr	Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Number of random values to be generated
<i>p</i>	FORTRAN 77: DOUBLE PRECISION Fortran 90: REAL(KIND=8), INTENT(IN) C: const double	Success probability <i>p</i> of a trial

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER (KIND=4), INTENT (OUT) C: int*	Vector of <i>n</i> Bernoulli distributed random values

Description

The vRngBernoulli function generates Bernoulli distributed random numbers with probability *p* of a single trial success, where

$$p \in R; 0 \leq p \leq 1.$$

A variate is called Bernoulli distributed, if after a trial it is equal to 1 with probability of success *p*, and to 0 with probability $1 - p$.

The probability distribution is given by:

$$P(X = 1) = p$$

$$P(X = 0) = 1 - p$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - p, & 0 \leq x < 1, x \in R. \\ 1, & x \geq 1 \end{cases}$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.

vRngGeometric

Generates geometrically distributed random values.

Syntax

Fortran:

```
status = virnggeometric( method, stream, n, r, p )
```

C:

```
status = viRngGeometric( method, stream, n, r, p );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Generation method. The specific value is as follows: VSL_RNG_METHOD_GEOOMETRIC_ICDF Inverse cumulative distribution function method.
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr	Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Number of random values to be generated
<i>p</i>	FORTRAN 77: DOUBLE PRECISION Fortran 90: REAL(KIND=8), INTENT(IN) C: const double	Success probability <i>p</i> of a trial

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(OUT) C: int*	Vector of <i>n</i> geometrically distributed random values

Description

The `vRngGeometric` function generates geometrically distributed random numbers with probability *p* of a single trial success, where $p \in R$; $0 < p < 1$.

A geometrically distributed variate represents the number of independent Bernoulli trials preceding the first success. The probability of a single Bernoulli trial success is p .

The probability distribution is given by:

$$P(X = k) = p \cdot (1 - p)^k, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_p(x) = \begin{cases} 0, & x < 0 \\ 1 - (1 - p)^{\lfloor x+1 \rfloor}, & 0 \geq x \end{cases} \quad x \in R.$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngBinomial

Generates binomially distributed random numbers.

Syntax

Fortran:

```
status = virngbinomial( method, stream, n, r, ntrial, p )
```

C:

```
status = viRngBinomial( method, stream, n, r, ntrial, p );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT</pre>	Generation method. The specific value is as follows: <code>VSL_RNG_METHOD_BINOMIAL_BTPE</code>
<i>stream</i>	<pre>FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr</pre>	<code>Fortran</code> : Descriptor of the stream state structure. <code>C</code> : Pointer to the stream state structure
<i>n</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT</pre>	Number of random values to be generated
<i>ntrial</i>	<pre>FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(IN) C: const int</pre>	Number of independent trials m
<i>p</i>	<pre>FORTRAN 77: DOUBLE PRECISION Fortran 90: REAL(KIND=8), INTENT(IN) C: const double</pre>	Success probability p of a single trial

Output Parameters

Name	Type	Description
<i>r</i>	<pre>FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(OUT) C: int*</pre>	Vector of n binomially distributed random values

Description

The `vRngBinomial` function generates binomially distributed random numbers with number of independent Bernoulli trials m , and with probability p of a single trial success, where $p \in R$; $0 \leq p \leq 1$, $m \in N$.

A binomially distributed variate represents the number of successes in m independent Bernoulli trials with probability of a single trial success p .

The probability distribution is given by:

$$P(X = k) = C_m^k p^k (1 - p)^{m-k}, k \in \{0, 1, \dots, m\}.$$

The cumulative distribution function is as follows:

$$F_{m,p}(x) = \begin{cases} 0, & x < 0 \\ \sum_{k=0}^{\lfloor x \rfloor} C_m^k p^k (1 - p)^{m-k}, & 0 \leq x < m, x \in R \\ 1, & x \geq m \end{cases}$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngHypergeometric

Generates hypergeometrically distributed random values.

Syntax

Fortran:

```
status = virnghypergeometric( method, stream, n, r, l, s, m )
```

C:

```
status = viRngHypergeometric( method, stream, n, r, l, s, m );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90

- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Generation method. The specific value is as follows: VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr	See brief description of the H2PE method in Table "Values of <method> in method parameter" Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT	Number of random values to be generated
<i>l</i>	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(IN) C: const int	Lot size l
<i>s</i>	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(IN) C: const int	Size of sampling without replacement s
<i>m</i>	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(IN) C: const int	Number of marked elements m

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(OUT) C: int*	Vector of <i>n</i> hypergeometrically distributed random values

Description

The `vRngHypergeometric` function generates hypergeometrically distributed random values with lot size l , size of sampling s , and number of marked elements in the lot m , where $l, m, s \in \mathbb{N} \cup \{0\}$; $l \geq \max(s, m)$.

Consider a lot of l elements comprising m "marked" and $l-m$ "unmarked" elements. A trial sampling without replacement of exactly s elements from this lot helps to define the hypergeometric distribution, which is the probability that the group of s elements contains exactly k marked elements.

The probability distribution is given by:

$$P(X = k) = \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}$$

, $k \in \{\max(0, s + m - l), \dots, \min(s, m)\}$

The cumulative distribution function is as follows:

$$F_{l,s,m}(x) = \begin{cases} 0, & x < \max(0, s + m - l) \\ \sum_{k=\max(0,s+m-l)}^{\lfloor x \rfloor} \frac{C_m^k C_{l-m}^{s-k}}{C_l^s}, & \max(0, s + m - l) \leq x \leq \min(s, m) \\ 1, & x > \min(s, m) \end{cases}$$

Return Values

<code>VSL_ERROR_OK</code> , <code>VSL_STATUS_OK</code>	Indicates no error, execution is successful.
<code>VSL_ERROR_NULL_PTR</code>	<i>stream</i> is a NULL pointer.
<code>VSL_RNG_ERROR_BAD_STREAM</code>	<i>stream</i> is not a valid random stream.
<code>VSL_RNG_ERROR_BAD_UPDATE</code>	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or $> nmax$.
<code>VSL_RNG_ERROR_NO_NUMBERS</code>	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
<code>VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED</code>	Period of the generator has been exceeded.
<code>VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED</code>	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngPoisson

Generates Poisson distributed random values.

Syntax

Fortran:

```
status = virngpoisson( method, stream, n, r, lambda )
```

C:

```
status = viRngPoisson( method, stream, n, r, lambda );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Generation method. The specific values are as follows: VSL_RNG_METHOD_POISSON_PTPE VSL_RNG_METHOD_POISSON_POISNORM
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	See brief description of the PTPE and POISNORM methods in Table "Values of <method> in method parameter" . Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Number of random values to be generated
<i>lambda</i>	FORTRAN 77: DOUBLE PRECISION Fortran 90: REAL(KIND=8), INTENT (IN) C: const double	Distribution parameter λ .

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT (OUT) C: int*	Vector of <i>n</i> Poisson distributed random values

Description

The vRngPoisson function generates Poisson distributed random numbers with distribution parameter λ , where $\lambda \in R$; $\lambda > 0$.

The probability distribution is given by:

...|::|::|::|::|::|::|...

$k \in \{0, 1, 2, \dots\}$.

The cumulative distribution function is as follows:

$$F_{\lambda}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, & x \geq 0, x \in R \\ 0, & x < 0 \end{cases}$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngPoissonV

Generates Poisson distributed random values with varying mean.

Syntax

Fortran:

```
status = virngpoissonv( method, stream, n, r, lambda )
```

C:

```
status = viRngPoissonV( method, stream, n, r, lambda );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>method</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT</pre>	Generation method. The specific value is as follows: VSL_RNG_METHOD_POISSONV_POISNORM
<i>stream</i>	<pre>FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT(IN) C: VSLStreamStatePtr</pre>	See brief description of the POISNORM method in Table "Values of <method> in <i>method</i> parameter" Fortran: Descriptor of the stream state structure. C: Pointer to the stream state structure
<i>n</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT(IN) C: const MKL_INT</pre>	Number of random values to be generated
<i>lambda</i>	<pre>FORTRAN 77: DOUBLE PRECISION Fortran 90: REAL(KIND=8), INTENT(IN) C: const double*</pre>	Array of <i>n</i> distribution parameters λ_i .

Output Parameters

Name	Type	Description
<i>r</i>	<pre>FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(OUT) C: int*</pre>	Vector of <i>n</i> Poisson distributed random values

Description

The vRngPoissonV function generates *n* Poisson distributed random numbers x_i ($i = 1, \dots, n$) with distribution parameter λ_i , where $\lambda_i \in R$; $\lambda_i > 0$.

The probability distribution is given by:

$$P(X_i = k) = \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, \quad k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_{\lambda_i}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k e^{-\lambda_i}}{k!}, & x \geq 0, x \in R \\ 0, & x < 0 \end{cases}$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED	Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

vRngNegBinomial

Generates random numbers with negative binomial distribution.

Syntax

Fortran:

```
status = virngnegbinomial( method, stream, n, r, a, p )
```

C:

```
status = viRngNegbinomial( method, stream, n, r, a, p );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
method	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Generation method. The specific value is: VSL_RNG_METHOD_NEGBINOMIAL_NBAR See brief description of the NBAR method in Table "Values of <method> in method parameter"

Name	Type	Description
<i>stream</i>	FORTRAN 77: INTEGER*4 stream(2) Fortran 90: TYPE (VSL_STREAM_STATE), INTENT (IN) C: VSLStreamStatePtr	Fortran: descriptor of the stream state structure. C: pointer to the stream state structure
<i>n</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, INTENT (IN) C: const MKL_INT	Number of random values to be generated
<i>a</i>	FORTRAN 77: DOUBLE PRECISION Fortran 90: REAL(KIND=8), INTENT (IN) C: const double	The first distribution parameter <i>a</i>
<i>p</i>	FORTRAN 77: DOUBLE PRECISION Fortran 90: REAL(KIND=8), INTENT (IN) C: const double	The second distribution parameter <i>p</i>

Output Parameters

Name	Type	Description
<i>r</i>	FORTRAN 77: INTEGER*4 Fortran 90: INTEGER(KIND=4), INTENT(OUT) C: int*	Vector of <i>n</i> random values with negative binomial distribution.

Description

The vRngNegBinomial function generates random numbers with negative binomial distribution and distribution parameters *a* and *p*, where *p*, *a* ∈ ℝ; $0 < p < 1$; $a > 0$.

If the first distribution parameter *a* ∈ ℤ, this distribution is the same as Pascal distribution. If *a* ∈ ℝ, the distribution can be interpreted as the expected time of *a*-th success in a sequence of Bernoulli trials, when the probability of success is *p*.

The probability distribution is given by:

$$P(X = k) = C_{a+k-1}^k p^a (1 - p)^k, k \in \{0, 1, 2, \dots\}.$$

The cumulative distribution function is as follows:

$$F_{\alpha, p}(x) = \begin{cases} \sum_{k=0}^{\lfloor x \rfloor} C_{\alpha+k-1}^k p^\alpha (1-p)^k, & x \geq 0 \\ 0, & x < 0 \end{cases}, x \in R$$

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR	<i>stream</i> is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM	<i>stream</i> is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS	Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED	Period of the generator has been exceeded.

Advanced Service Routines

This section describes service routines for registering a user-designed basic generator (`vslRegisterBrng`) and for obtaining properties of the previously registered basic generators (`vslGetBrngProperties`). See [VSL Notes](#) ("Basic Generators" section of VSL Structure chapter) for substantiation of the need for several basic generators including user-defined BRNGs.

NOTE

The `vslRegisterBrng` function is provided in C for registering a user-defined basic generator, but it is not supported for Fortran. If you need to use a user-defined generator in Fortran, use an abstract basic random generator and abstract stream as described in the VSL Notes.

Data types

The Advanced Service routines refer to a structure defining the properties of the basic generator.

This structure is described in Fortran 90 as follows:

```
TYPE VSL_BRNG_PROPERTIES
    INTEGER streamstatesize
    INTEGER nseeds
    INTEGER includeszero
    INTEGER wordsize
    INTEGER nbBits
    INTEGER reserved(8)
END TYPE VSL_BRNG_PROPERTIES
```

This structure is described in C as follows:

```
typedef struct _VSLBRngProperties {
    int StreamStateSize;
    int NSeeds;
```

```

int IncludesZero;
int WordSize;
int NBits;
InitStreamPtr InitStream;
sBRngPtr sBRng;
dBRngPtr dBRng;
iBRngPtr iBRng;
} VSLBRngProperties;

```

The following table provides brief descriptions of the fields engaged in the above structure:

Field Descriptions

Field	Short Description
Fortran: streamstatesize C: StreamStateSize	The size, in bytes, of the stream state structure for a given basic generator.
Fortran: nseeds C: NSeeds	The number of 32-bit initial conditions (seeds) necessary to initialize the stream state structure for a given basic generator.
Fortran: includeszero C: IncludesZero	Flag value indicating whether the generator can produce a random 0.
Fortran: wordsize C: WordSize	Machine word size, in bytes, used in integer-value computations. Possible values: 4, 8, and 16 for 32, 64, and 128-bit generators, respectively.
Fortran: nbits C: NBits	The number of bits required to represent a random value in integer arithmetic. Note that, for instance, 48-bit random values are stored to 64-bit (8 byte) memory locations. In this case, wordsize/WordSize is equal to 8 (number of bytes used to store the random value), while nbits/NBits contains the actual number of bits occupied by the value (in this example, 48).
C: InitStream	Contains the pointer to the initialization routine of a given basic generator.
C: sBRng	Contains the pointer to the basic generator of single precision real numbers uniformly distributed over the interval (a,b) (float).
C: dBRng	Contains the pointer to the basic generator of double precision real numbers uniformly distributed over the interval (a,b) (double).
C: iBRng	Contains the pointer to the basic generator of integer numbers with uniform bit distribution (unsigned int).
Fortran: reserved(8)	Reserved for internal use.

NOTE

Using Advanced Service routines for defining generators is not supported for Fortran, but you can use the Fortran interface to get information about a previously-registered generator using [vslGetBrngProperties](#).

vslRegisterBrng

Registers user-defined basic generator.

¹ A specific generator that permits operations over single bits and bit groups of random numbers.

Syntax

C:

```
brng = vslRegisterBrng( &properties );
```

Include Files

- C: mkl.h

Input Parameters

Name	Type	Description
<i>properties</i> C:	const VSLBRngProperties*	Pointer to the structure containing properties of the basic generator to be registered

NOTE

This function is not supported for Fortran. If you need to use a user-defined generator in Fortran, use an abstract basic random generator and abstract stream as described in the VSL Notes.

Output Parameters

Name	Type	Description
<i>brng</i>	C: int	Number (index) of the registered basic generator; used for identification. Negative values indicate the registration error.

Description

An example of a registration procedure can be found in the respective directory of the VSL examples.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_BRNG_TABLE_FULL	Registration cannot be completed due to lack of free entries in the table of registered BRNGs.
VSL_RNG_ERROR_BAD_STREAM_STATE_SIZE	Bad value in StreamStateSize field.
VSL_RNG_ERROR_BAD_WORD_SIZE	Bad value in WordSize field.
VSL_RNG_ERROR_BAD_NSEEDS	Bad value in NSeeds field.
VSL_RNG_ERROR_BAD_NBITS	Bad value in NBits field.
VSL_ERROR_NULL_PTR	At least one of the fields iBrng, dBrng, sBrng or InitStream is a NULL pointer.

vslGetBrngProperties

Returns structure with properties of a given basic generator.

Syntax

Fortran:

```
status = vslgetbrngproperties( brng, properties )
```

C:

```
status = vslGetBrngProperties( brng, &properties );
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>brng</i>	Fortran: INTEGER(KIND=4), INTENT(IN) C: const int	Number (index) of the registered basic generator; used for identification. See specific values in Table "Values of <i>brng</i> parameter" . Negative values indicate the registration error.

Output Parameters

Name	Type	Description
<i>properties</i>	Fortran: TYPE(VSL_BRNG_PROPERTIES), INTENT(OUT) C: VSLBRngProperties*	Pointer to the structure containing properties of the generator with number <i>brng</i>

Description

The `vslGetBrngProperties` function returns a structure with properties of a given basic generator.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK	Indicates no error, execution is successful.
VSL_RNG_ERROR_INVALID_BRNG_INDEX	BRNG index is invalid.

Formats for User-Designed Generators

To register a user-designed basic generator using `vslRegisterBrng` function, you need to pass the pointer `iBrng` to the integer-value implementation of the generator; the pointers `sBrng` and `dBrng` to the generator implementations for single and double precision values, respectively; and pass the pointer `InitStream` to the stream initialization routine. See recommendations below on defining such functions with input and output arguments. An example of the registration procedure for a user-designed generator can be found in the respective directory of VSL examples.

NOTE

This function is not supported for Fortran. If you need to use a user-defined generator in Fortran, use an abstract basic random generator and abstract stream as described in the VSL Notes.

The respective pointers are defined as follows:

```
typedef int(*InitStreamPtr)( int method, VSLStreamStatePtr stream, int n, const unsigned int params[] );
typedef int(*sBRngPtr)( VSLStreamStatePtr stream, int n, float r[], float a, float b );
typedef int(*dBRngPtr)( VSLStreamStatePtr stream, int n, double r[], double a, double b );
typedef int(*iBRngPtr)( VSLStreamStatePtr stream, int n, unsigned int r[] );
```

InitStream

C:

```
int MyBrngInitStream( int method, VSLStreamStatePtr stream, int n, const unsigned int params[] )
{
    /* Initialize the stream */
    ...
} /* MyBrngInitStream */
```

Description

The initialization routine of a user-designed generator must initialize *stream* according to the specified initialization *method*, initial conditions *params* and the argument *n*. The value of *method* determines the initialization method to be used.

- If *method* is equal to 1, the initialization is by the standard generation method, which must be supported by all basic generators. In this case the function assumes that the *stream* structure was not previously initialized. The value of *n* is used as the actual number of 32-bit values passed as initial conditions through *params*. Note, that the situation when the actual number of initial conditions passed to the function is not sufficient to initialize the generator is not an error. Whenever it occurs, the basic generator must initialize the missing conditions using default settings.
- If *method* is equal to 2, the generation is by the leapfrog method, where *n* specifies the number of computational nodes (independent streams). Here the function assumes that the *stream* was previously initialized by the standard generation method. In this case *params* contains only one element, which identifies the computational node. If the generator does not support the leapfrog method, the function must return the error code `VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED`.
- If *method* is equal to 3, the generation is by the block-splitting method. Same as above, the *stream* is assumed to be previously initialized by the standard generation method; *params* is not used, *n* identifies the number of skipped elements. If the generator does not support the block-splitting method, the function must return the error code `VSL_RNG_ERROR_SKIPHEAD_UNSUPPORTED`.

For a more detailed description of the leapfrog and the block-splitting methods, refer to the description of `vslLeapfrogStream` and `vslSkipAheadStream`, respectively.

Stream state structure is individual for every generator. However, each structure has a number of fields that are the same for all the generators:

C:

```
typedef struct
{
    unsigned int Reserved1[2];
    unsigned int Reserved2[2];
    [fields specific for the given generator]
} MyStreamState;
```

The fields *Reserved1* and *Reserved2* are reserved for private needs only, and must not be modified by the user. When including specific fields into the structure, follow the rules below:

- The fields must fully describe the current state of the generator. For example, the state of a linear congruential generator can be identified by only one initial condition;

- If the generator can use both the leapfrog and the block-splitting methods, additional fields should be introduced to identify the independent streams. For example, in $LCG(a, c, m)$, apart from the initial conditions, two more fields should be specified: the value of the multiplier a^k and the value of the increment $(a^k - 1)c / (a - 1)$.

For a more detailed discussion, refer to [Knuth81], and [Gentle98]. An example of the registration procedure can be found in the respective directory of VSL examples.

iBRng

C:

```
int iMyBrng( VSLStreamStatePtr stream, int n, unsigned int r[] )
{
    int i; /* Loop variable */
    /* Generating integer random numbers */
    /* Pay attention to word size needed to
     * store only random number */
    for( i = 0; i < n; i++)
    {
        r[i] = ...;
    }
    /* Update stream state */
    ...
    return errcode;
} /* iMyBrng */
```

NOTE

When using 64 and 128-bit generators, consider digit capacity to store the numbers to the random vector r correctly. For example, storing one 64-bit value requires two elements of r , the first to store the lower 32 bits and the second to store the higher 32 bits. Similarly, use 4 elements of r to store a 128-bit value.

sBRng

C:

```
int sMyBrng( VSLStreamStatePtr stream, int n, float r[], float a, float b )
{
    int i; /* Loop variable */
    /* Generating float (a,b) random numbers */
    for ( i = 0; i < n; i++ )
    {
        r[i] = ...;
    }
    /* Update stream state */
    ...
    return errcode;
} /* sMyBrng */
```

dB Rng

C:

```
int dMyBrng( VSLStreamStatePtr stream, int n, double r[], double a, double b )
{
    int i; /* Loop variable */
    /* Generating double (a,b) random numbers */
    for ( i = 0; i < n; i++ )
    {
        r[i] = ...;
```

```

    }
    /* Update stream state */
    ...
    return errcode;
} /* dMyBrng */

```

Convolution and Correlation

Intel MKL VSL provides a set of routines intended to perform linear convolution and correlation transformations for single and double precision real and complex data.

For correct definition of implemented operations, see the [Mathematical Notation and Definitions](#) section.

The current implementation provides:

- Fourier algorithms for one-dimensional single and double precision real and complex data
- Fourier algorithms for multi-dimensional single and double precision real and complex data
- Direct algorithms for one-dimensional single and double precision real and complex data
- Direct algorithms for multi-dimensional single and double precision real and complex data

One-dimensional algorithms cover the following functions from the IBM* ESSL library:

`SCONF, SCORF`

`SCOND, SCORD`

`SDCON, SDCOR`

`DDCON, DDCOR`

`SDDCON, SDDCOR.`

Special wrappers are designed to simulate these ESSL functions. The wrappers are provided as sample sources for Fortran and C:

`$(MKL)/examples/vslc/essl/vsl_wrappers`

`$(MKL)/examples/vslf/essl/vsl_wrappers`

Additionally, you can browse the examples demonstrating the calculation of the ESSL functions through the wrappers:

`$(MKL)/examples/vslc/essl`

`$(MKL)/examples/vslf/essl`

The convolution and correlation API provides interfaces for FORTRAN 77, Fortran 90 and C/89 languages. You can use the C89 interface with later versions of the C/C++. You can use the Fortran 90 interface with programs written in Fortran 95.

Intel MKL provides the `mkl_vsl.h` header file for C, and the `mkl_vsl.f90` and `mkl_vsl.f77` header files for Fortran. All header files are in the directory

`$(MKL)/include`

See more details about the Fortran header in [Random Number Generators](#) section of this chapter.

The convolution and correlation API is implemented through task objects, or tasks. Task object is a data structure, or descriptor, which holds parameters that determine the specific convolution or correlation operation. Such parameters may be precision, type, and number of dimensions of user data, an identifier of the computation algorithm to be used, shapes of data arrays, and so on.

All the Intel MKL VSL convolution and correlation routines process task objects in one way or another: either create a new task descriptor, change the parameter settings, compute mathematical results of the convolution or correlation using the stored parameters, or perform other operations. Accordingly, all routines are split into the following groups:

Task Constructors - routines that create a new task object descriptor and set up most common parameters.

Task Editors - routines that can set or modify some parameter settings in the existing task descriptor.

Task Execution Routines - compute results of the convolution or correlation operation over the actual input data, using the operation parameters held in the task descriptor.

Task Copy - routines used to make several copies of the task descriptor.

Task Destructors - routines that delete task objects and free the memory.

When the task is executed or copied for the first time, a special process runs which is called task commitment. During this process, consistency of task parameters is checked and the required work data are prepared. If the parameters are consistent, the task is tagged as committed successfully. The task remains committed until you edit its parameters. Hence, the task can be executed multiple times after a single commitment process. Since the task commitment process may include costly intermediate calculations such as preparation of Fourier transform of input data, launching the process only once can help speed up overall performance.

Naming Conventions

The names of Fortran routines in the convolution and correlation API are written in lowercase (`vslsconvexec`), while the names of Fortran types and constants are written in uppercase. The names are not case-sensitive.

The names of C routines, types, and constants in the convolution and correlation API are case-sensitive and can contain both lowercase and uppercase characters (`vslsConvExec`).

The names of routines have the following structure:

`vsl[datatype] {Conv|Corr}<base name>` for the C interface

`vsl[datatype] {conv|corr}<base name>` for the Fortran interface

where

- `vsl` is a prefix indicating that the routine belongs to Vector Statistical Library of Intel® MKL.
- `[datatype]` is optional. If present, the symbol specifies the type of the input and output data and can be `s` (for single precision real type), `d` (for double precision real type), `c` (for single precision complex type), or `z` (for double precision complex type).
- `Conv` or `Corr` specifies whether the routine refers to convolution or correlation task, respectively.
- `<base name>` field specifies a particular functionality that the routine is designed for, for example, `NewTask`, `DeleteTask`.

Data Types

All convolution or correlation routines use the following types for specifying data objects:

Type	Data Object
FORTRAN 77: <code>INTEGER*4 task (2)</code>	Pointer to a task descriptor for convolution
Fortran 90: <code>TYPE(VSL_CONV_TASK)</code>	
C: <code>VSLConvTaskPtr</code>	
FORTRAN 77: <code>INTEGER*4 task (2)</code>	Pointer to a task descriptor for correlation
Fortran 90: <code>TYPE(VSL_CORR_TASK)</code>	
C: <code>VSLCorrTaskPtr</code>	
FORTRAN 77: <code>REAL*4</code>	Input/output user real data in single precision
Fortran 90: <code>REAL(KIND=4)</code>	

Type	Data Object
C: float	
FORTRAN 77: REAL*8	Input/output user real data in double precision
Fortran 90: REAL (KIND=8)	
C: double	
FORTRAN 77: COMPLEX*8	Input/output user complex data in single precision
Fortran 90: COMPLEX (KIND=4)	
C: MKL_Complex8	
FORTRAN 77: COMPLEX*16	Input/output user complex data in double precision
Fortran 90: COMPLEX (KIND=8)	
C: MKL_Complex16	
FORTRAN 77: INTEGER	All other data
Fortran 90: INTEGER	
C: int	

Generic integer type (without specifying the byte size) is used for all integer data.

NOTE

The actual size of the generic integer type is platform-dependent. Before you compile your application, set an appropriate byte size for integers. See details in the 'Using the ILP64 Interface vs. LP64 Interface' section of the *Intel® MKL User's Guide*.

Parameters

Basic parameters held by the task descriptor are assigned values when the task object is created, copied, or modified by task editors. Parameters of the correlation or convolution task are initially set up by task constructors when the task object is created. Parameter changes or additional settings are made by task editors. More parameters which define location of the data being convolved need to be specified when the task execution routine is invoked.

According to how the parameters are passed or assigned values, all of them can be categorized as either explicit (directly passed as routine parameters when a task object is created or executed) or optional (assigned some default or implicit values during task construction).

The following table lists all applicable parameters used in the Intel MKL convolution and correlation API.

Convolution and Correlation Task Parameters

Name	Category	Type	Default Value Label	Description
<i>job</i>	explicit	integer	Implied by the constructor name	Specifies whether the task relates to convolution or correlation
<i>type</i>	explicit	integer	Implied by the constructor name	Specifies the type (real or complex) of the input/output data. Set to real in the current version.
<i>precision</i>	explicit	integer	Implied by the constructor name	Specifies precision (single or double) of the input/output data to be provided in arrays <i>x,y,z</i> .

Name	Category	Type	Default Value Label	Description
<i>mode</i>	explicit	integer	None	Specifies whether the convolution/correlation computation should be done via Fourier transforms, or by a direct method, or by automatically choosing between the two. See SetMode for the list of named constants for this parameter.
<i>method</i>	optional	integer	"auto"	Hints at a particular computation method if several methods are available for the given <i>mode</i> . Setting this parameter to "auto" means that software will choose the best available method.
<i>internal_precision</i>	optional	integer	Set equal to the value of <i>precision</i>	Specifies precision of internal calculations. Can enforce double precision calculations even when input/output data are single precision. See SetInternalPrecision for the list of named constants for this parameter.
<i>dims</i>	explicit	integer	None	Specifies the rank (number of dimensions) of the user data provided in arrays <i>x,y,z</i> . Can be in the range from 1 to 7.
<i>x,y</i>	explicit	real arrays	None	Specify input data arrays. See Data Allocation for more information.
<i>z</i>	explicit	real array	None	Specifies output data array. See Data Allocation for more information.
<i>xshape,</i> <i>yshape,</i> <i>zshape</i>	explicit	integer arrays	None	Define shapes of the arrays <i>x, y, z</i> . See Data Allocation for more information.
<i>xstride,</i> <i>ystride,</i> <i>zstride</i>	explicit	integer arrays	None	Define strides within arrays <i>x, y, z</i> , that is specify the physical location of the input and output data in these arrays. See Data Allocation for more information.
<i>start</i>	optional	integer array	Undefined	Defines the first element of the mathematical result that will be stored to output array <i>z</i> . See SetStart and Data Allocation for more information.
<i>decimation</i>	optional	integer array	Undefined	Defines how to thin out the mathematical result that will be stored to output array <i>z</i> . See SetDecimation and Data Allocation for more information.

Users of the C or C++ language may pass the NULL pointer instead of either or all of the parameters *xstride*, *ystride*, or *zstride* for multi-dimensional calculations. In this case, the software assumes the dense data allocation for the arrays *x*, *y*, or *z* due to the Fortran-style "by columns" representation of multi-dimensional arrays.

Task Status and Error Reporting

The task status is an integer value, which is zero if no error has been detected while processing the task, or a specific non-zero error code otherwise. Negative status values indicate errors, and positive values indicate warnings.

An error can be caused by invalid parameter values, a system fault like a memory allocation failure, or can be an internal error self-detected by the software.

Each task descriptor contains the current status of the task. When creating a task object, the constructor assigns the `VSL_STATUS_OK` status to the task. When processing the task afterwards, other routines such as editors or executors can change the task status if an error occurs and write a corresponding error code into the task status field.

Note that at the stage of creating a task or editing its parameters, the set of parameters may be inconsistent. The parameter consistency check is only performed during the task commitment operation, which is implicitly invoked before task execution or task copying. If an error is detected at this stage, task execution or task copying is terminated and the task descriptor saves the corresponding error code. Once an error occurs, any further attempts to process that task descriptor is terminated and the task keeps the same error code.

Normally, every convolution or correlation function (except `DeleteTask`) returns the status assigned to the task while performing the function operation.

The header files define symbolic names for the status codes. These names for the C/C++ interface are defined as macros via the `#define` statements. These names for the Fortran interface are defined as integer constants via the `PARAMETER` operators.

If there is no error, the `VSL_STATUS_OK` status is returned, which is defined as zero:

```
C/C++:      #define VSL_STATUS_OK 0
F90/F95:    INTEGER(KIND=4) VSL_STATUS_OK
            PARAMETER(VSL_STATUS_OK = 0)
F77:        INTEGER*4 VSL_STATUS_OK
            PARAMETER(VSL_STATUS_OK = 0)
```

In case of an error, a non-zero error code is returned, which indicates the origin of the failure. The following status codes for the convolution/correlation error codes are pre-defined in the header files.

Convolution/Correlation Status Codes

Status Code	Description
<code>VSL_CC_ERROR_NOT_IMPLEMENTED</code>	Requested functionality is not implemented.
<code>VSL_CC_ERROR_ALLOCATION_FAILURE</code>	Memory allocation failure.
<code>VSL_CC_ERROR_BAD_DESCRIPTOR</code>	Task descriptor is corrupted.
<code>VSL_CC_ERROR_SERVICE_FAILURE</code>	A service function has failed.
<code>VSL_CC_ERROR_EDIT_FAILURE</code>	Failure while editing the task.
<code>VSL_CC_ERROR_EDIT_PROHIBITED</code>	You cannot edit this parameter.
<code>VSL_CC_ERROR_COMMIT_FAILURE</code>	Task commitment has failed.
<code>VSL_CC_ERROR_COPY_FAILURE</code>	Failure while copying the task.
<code>VSL_CC_ERROR_DELETE_FAILURE</code>	Failure while deleting the task.
<code>VSL_CC_ERROR_BAD_ARGUMENT</code>	Bad argument or task parameter.
<code>VSL_CC_ERROR_JOB</code>	Bad parameter: <i>job</i> .
<code>SL_CC_ERROR_KIND</code>	Bad parameter: <i>kind</i> .
<code>VSL_CC_ERROR_MODE</code>	Bad parameter: <i>mode</i> .
<code>VSL_CC_ERROR_METHOD</code>	Bad parameter: <i>method</i> .

Status Code	Description
VSL_CC_ERROR_TYPE	Bad parameter: <i>type</i> .
VSL_CC_ERROR_EXTERNAL_PRECISION	Bad parameter: <i>external_precision</i> .
VSL_CC_ERROR_INTERNAL_PRECISION	Bad parameter: <i>internal_precision</i> .
VSL_CC_ERROR_PRECISION	Incompatible external/internal precisions.
VSL_CC_ERROR_DIMS	Bad parameter: <i>dims</i> .
VSL_CC_ERROR_XSHAPE	Bad parameter: <i>xshape</i> .
VSL_CC_ERROR_YSHAPE	Bad parameter: <i>yshape</i> .
VSL_CC_ERROR_ZSHAPE	Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > <i>nmax</i> .
VSL_CC_ERROR_XSTRIDE	Bad parameter: <i>xstride</i> .
VSL_CC_ERROR_YSTRIDE	Bad parameter: <i>ystride</i> .
VSL_CC_ERROR_ZSTRIDE	Bad parameter: <i>zstride</i> .
VSL_CC_ERROR_X	Bad parameter: <i>x</i> .
VSL_CC_ERROR_Y	Bad parameter: <i>y</i> .
VSL_CC_ERROR_Z	Bad parameter: <i>z</i> .
VSL_CC_ERROR_START	Bad parameter: <i>start</i> .
VSL_CC_ERROR_DECIMATION	Bad parameter: <i>decimation</i> .
VSL_CC_ERROR_OTHER	Another error.

Task Constructors

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters. No additional parameter adjustment is typically required and other routines can use the task object.

Intel® MKL implementation of the convolution and correlation API provides two different forms of constructors: a general form and an X-form. X-form constructors work in the same way as the general form constructors but also assign particular data to the first operand vector used in the convolution or correlation operation (stored in array *x*).

Using X-form constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector held in array *x* against different vectors held in array *y*. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Each constructor routine has an associated one-dimensional version that provides algorithmic and computational benefits.

NOTE

If the constructor fails to create a task descriptor, it returns the NULL task pointer.

The Table "Task Constructors" lists available task constructors:

Task Constructors

Routine	Description
<code>vslConvNewTask/vslCorrNewTask</code>	Creates a new convolution or correlation task descriptor for a multidimensional case.
<code>vslConvNewTask1D/vslCorrNewTask1D</code>	Creates a new convolution or correlation task descriptor for a one-dimensional case.
<code>vslConvNewTaskX/vslCorrNewTaskX</code>	Creates a new convolution or correlation task descriptor as an X-form for a multidimensional case.
<code>vslConvNewTaskX1D/vslCorrNewTaskX1D</code>	Creates a new convolution or correlation task descriptor as an X-form for a one-dimensional case.

`vslConvNewTask/vslCorrNewTask`

Creates a new convolution or correlation task descriptor for multidimensional case.

Syntax**Fortran:**

```
status = vslsconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vsldconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslcconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslzconvnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslscorrnewtask(task, mode, dims, xshape, yshape, zshape)
status = vsldcorrnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslccorrnewtask(task, mode, dims, xshape, yshape, zshape)
status = vslzcorrnewtask(task, mode, dims, xshape, yshape, zshape)
```

C:

```
status = vslsConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vsldConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslcConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslzConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslsCorrNewTask(task, mode, dims, xshape, yshape, zshape);
status = vsldCorrNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslcCorrNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslzCorrNewTask(task, mode, dims, xshape, yshape, zshape);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>mode</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER C: const MKL_INT	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
<i>dims</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER C: const MKL_INT	Rank of user data. Specifies number of dimensions for the input and output arrays <i>x</i> , <i>y</i> , and <i>z</i> used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.
<i>xshape</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION(*) C: const int[]	Defines the shape of the input data for the source array <i>x</i> . See Data Allocation for more information.
<i>yshape</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION(*) C: const int[]	Defines the shape of the input data for the source array <i>y</i> . See Data Allocation for more information.
<i>zshape</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION(*) C: const int[]	Defines the shape of the output data to be stored in array <i>z</i> . See Data Allocation for more information.

Output Parameters

Name	Type	Description
<i>task</i>	FORTRAN 77: INTEGER*4 task(2) for vslsconvnewtask, vslldconvnewtask, vslccconvnewtask, vslzconvnewtask Fortran 90: TYPE(VSL_CONV_TASK) for vslsconvnewtask, vslldconvnewtask, vslccconvnewtask, vslzconvnewtask	Pointer to the task descriptor if created successfully or NULL pointer otherwise.

Name	Type	Description
	TYPE(VSL_CORR_TASK) for vslscorrnewtask, vsldcorrnewtask, vslccorrnewtask, vslzcorrnewtask	
	C: VSLConvTaskPtr* for vslsConvNewTask, vsldConvNewTask, vslcConvNewTask, vslzConvNewTask	
	VSLCorrTaskPtr* for vslsCorrNewTask, vsldCorrNewTask, vslcCorrNewTask, vslzCorrNewTask	
status	FORTRAN 77: INTEGER Fortran 90: INTEGER C: int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

Description

Each vslConvNewTask/vslCorrNewTask constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)).

The parameters *xshape*, *yshape*, and *zshape* define the shapes of the input and output data provided by the arrays *x*, *y*, and *z*, respectively. Each shape parameter is an array of integers with its length equal to the value of *dims*. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter *dims* is 1, then *xshape*, *yshape*, *zshape* are equal to the number of elements read from the arrays *x* and *y* or stored to the array *z*. Note that values of shape parameters may differ from physical shapes of arrays *x*, *y*, and *z* if non-trivial strides are assigned.

If the constructor fails to create a task descriptor, it returns a NULL task pointer.

vslConvNewTask1D/vslCorrNewTask1D

Creates a new convolution or correlation task descriptor for one-dimensional case.

Syntax

Fortran:

```
status = vslsconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vsldconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vslcconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vslzconvnewtask1d(task, mode, xshape, yshape, zshape)
status = vslscorrnewtask1d(task, mode, xshape, yshape, zshape)
status = vsldcorrnewtask1d(task, mode, xshape, yshape, zshape)
status = vslccorrnewtask1d(task, mode, xshape, yshape, zshape)
```

```
status = vslzcorrnewtask1d(task, mode, xshape, yshape, zshape)
```

C:

```
status = vslsConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vsldConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vslcConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vslzConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vslsCorrNewTask1D(task, mode, xshape, yshape, zshape);
status = vsldCorrNewTask1D(task, mode, xshape, yshape, zshape);
status = vslcCorrNewTask1D(task, mode, xshape, yshape, zshape);
status = vslzCorrNewTask1D(task, mode, xshape, yshape, zshape);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
mode	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER <small>C:</small> const MKL_INT	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
xshape	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER <small>C:</small> const MKL_INT	Defines the length of the input data sequence for the source array x. See Data Allocation for more information.
yshape	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER <small>C:</small> const MKL_INT	Defines the length of the input data sequence for the source array y. See Data Allocation for more information.
zshape	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER <small>C:</small> const MKL_INT	Defines the length of the output data sequence to be stored in array z. See Data Allocation for more information.

Output Parameters

Name	Type	Description
task	<small>FORTRAN 77:</small> <small>INTEGER*4 task(2) for</small> <small>vslsconvnewtask1d,</small> <small>vsldconvnewtask1d,</small> <small>vslcconvnewtask1d,</small> <small>vslzconvnewtask1d</small>	Pointer to the task descriptor if created successfully or NULL pointer otherwise.

Name	Type	Description
	<pre>INTEGER*4 task(2) for vslscorrnewtask1d, vsldcorrnewtask1d, vslccorrnewtask1d, vslzcorrnewtask1d Fortran 90: TYPE(VSL_CONV_TASK) for vslsconvnewtask1d, vsldcconvnewtask1d, vslccconvnewtask1d, vslzconvnewtask1d TYPE(VSL_CORR_TASK) for vslscorrnewtask1d, vsldcorrnewtask1d, vslccorrnewtask1d, vslzcorrnewtask1d C: VSLConvTaskPtr* for vslsConvNewTask1D, vsldConvNewTask1D, vslcConvNewTask1D, vslzConvNewTask1D VSLCorrTaskPtr* for vslsCorrNewTask1D, vsldCorrNewTask1D, vslcCorrNewTask1D, vslzCorrNewTask1D</pre>	
<i>status</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER C: int	Set to <code>VSL_STATUS_OK</code> if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTask1D/vslCorrNewTask1D` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)). Unlike `vslConvNewTask/vslCorrNewTask`, these routines represent a special one-dimensional version of the constructor which assumes that the value of the parameter `dims` is 1. The parameters `xshape`, `yshape`, and `zshape` are equal to the number of elements read from the arrays `x` and `y` or stored to the array `z`. You explicitly assign the shape parameters when calling the constructor.

vslConvNewTaskX/vslCorrNewTaskX

Creates a new convolution or correlation task descriptor for multidimensional case and assigns source data to the first operand vector.

Syntax

Fortran:

```
status = vslsconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
```

```

status = vsldconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslcconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslzconvnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslscorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vsldcorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslccorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)
status = vslzcorrnewtaskx(task, mode, dims, xshape, yshape, zshape, x, xstride)

```

C:

```

status = vslsConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vsldConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslcConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslzConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslsCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vsldCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslcCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslzCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);

```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
mode	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER <small>C:</small> const MKL_INT	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
dims	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER <small>C:</small> const MKL_INT	Rank of user data. Specifies number of dimensions for the input and output arrays <i>x</i> , <i>y</i> , and <i>z</i> used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.
xshape	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER, DIMENSION(*) <small>C:</small> const int[]	Defines the shape of the input data for the source array <i>x</i> . See Data Allocation for more information.
yshape	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER, DIMENSION(*) <small>C:</small> const int[]	Defines the shape of the input data for the source array <i>y</i> . See Data Allocation for more information.

Name	Type	Description
<code>zshape</code>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER, DIMENSION(*)</p> <p>C: const int[]</p>	Defines the shape of the output data to be stored in array <code>z</code> . See Data Allocation for more information.
<code>x</code>	<p>FORTRAN 77: REAL*4 for real data in single precision flavors, REAL*8 for real data in double precision flavors, COMPLEX*8 for complex data in single precision flavors, COMPLEX*16 for complex data in double precision flavors</p> <p>Fortran 90: REAL(KIND=4), DIMENSION (*) for real data in single precision flavors, REAL(KIND=8), DIMENSION (*) for real data in double precision flavors, COMPLEX(KIND=4), DIMENSION (*) for complex data in single precision flavors, COMPLEX(KIND=8), DIMENSION (*) for complex data in double precision flavors</p> <p>C: const float[] for real data in single precision flavors, const double[] for real data in double precision flavors, const MKL_Complex8[] for complex data in single precision flavors, const MKL_Complex16[] for complex data in double precision flavors</p>	Pointer to the array containing input data for the first operand vector. See Data Allocation for more information.
<code>xstride</code>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER, DIMENSION(*)</p> <p>C: const int[]</p>	Strides for input data in the array <code>x</code> .

Output Parameters

Name	Type	Description
<i>task</i>	<pre> FORTRAN 77: INTEGER*4 task(2) for vslsconvnewtaskx, vsldconvnewtaskx, vslccconvnewtaskx, vslzconvnewtaskx INTEGER*4 task(2) for vslscorrnewtaskx, vsldcorrnewtaskx, vslccorrnewtaskx, vslzcorrnewtaskx Fortran 90: TYPE(VSL_CONV_TASK) for vslsconvnewtaskx, vsldconvnewtaskx, vslccconvnewtaskx, vslzconvnewtaskx TYPE(VSL_CORR_TASK) for vslscorrnewtaskx, vsldcorrnewtaskx, vslccorrnewtaskx, vslzcorrnewtaskx C: VSLConvTaskPtr* for vslsConvNewTaskX, vsldConvNewTaskX, vslcConvNewTaskX, vslzConvNewTaskX VSLCorrTaskPtr* for vslsCorrNewTaskX, vsldCorrNewTaskX, vslcCorrNewTaskX, vslzCorrNewTaskX </pre>	Pointer to the task descriptor if created successfully or <code>NULL</code> pointer otherwise.
<i>status</i>	<pre> FORTRAN 77: INTEGER Fortran 90: INTEGER C: int </pre>	Set to <code>VSL_STATUS_OK</code> if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTaskX/vslCorrNewTaskX` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)).

Unlike `vslConvNewTask/vslCorrNewTask`, these routines represent the so called X-form version of the constructor, which means that in addition to creating the task descriptor they assign particular data to the first operand vector in array *x* used in convolution or correlation operation. The task descriptor created by

the `vslConvNewTaskX/vslCorrNewTaskX` constructor keeps the pointer to the array `x` all the time, that is, until the task object is deleted by one of the destructor routines (see `vslConvDeleteTask/vslCorrDeleteTask`).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array `x` against different vectors in array `y`. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters `xshape`, `yshape`, and `zshape` define the shapes of the input and output data provided by the arrays `x`, `y`, and `z`, respectively. Each shape parameter is an array of integers with its length equal to the value of `dims`. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter `dims` is 1, then `xshape`, `yshape`, and `zshape` are equal to the number of elements read from the arrays `x` and `y` or stored to the array `z`. Note that values of shape parameters may differ from physical shapes of arrays `x`, `y`, and `z` if non-trivial strides are assigned.

The stride parameter `xstride` specifies the physical location of the input data in the array `x`. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter `xstride` is `s`, then only every s^{th} element of the array `x` will be used to form the input sequence. The stride value must be positive or negative but not zero.

`vslConvNewTaskX1D/vslCorrNewTaskX1D`

Creates a new convolution or correlation task descriptor for one-dimensional case and assigns source data to the first operand vector.

Syntax

Fortran:

```
status = vslsconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vsldconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslcconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslzconvnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslscorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vsldcorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslccorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
status = vslzcorrnewtaskx1d(task, mode, xshape, yshape, zshape, x, xstride)
```

C:

```
status = vslsConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vsldConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslcConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslzConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslsCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vsldCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslcCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslzCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
mode	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER <small>C:</small> const MKL_INT	Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table "Values of mode parameter" for a list of possible values.
xshape	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER <small>C:</small> const MKL_INT	Defines the length of the input data sequence for the source array x. See Data Allocation for more information.
yshape	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER <small>C:</small> const MKL_INT	Defines the length of the input data sequence for the source array y. See Data Allocation for more information.
zshape	<small>FORTRAN 77:</small> INTEGER <small>Fortran 90:</small> INTEGER <small>C:</small> const MKL_INT	Defines the length of the output data sequence to be stored in array z. See Data Allocation for more information.
x	<small>FORTRAN 77:</small> REAL*4 for real data in single precision flavors, <small>REAL*8</small> for real data in double precision flavors, <small>COMPLEX*8</small> for complex data in single precision flavors, <small>COMPLEX*16</small> for complex data in double precision flavors <small>Fortran 90:</small> <small>REAL(KIND=4), DIMENSION (*)</small> for real data in single precision flavors, <small>REAL(KIND=8), DIMENSION (*)</small> for real data in double precision flavors, <small>COMPLEX(KIND=4), DIMENSION (*)</small> for complex data in single precision flavors, <small>COMPLEX(KIND=8), DIMENSION (*)</small> for complex data in double precision flavors <small>C:</small> const float[] for real data in single precision flavors,	Pointer to the array containing input data for the first operand vector. See Data Allocation for more information.

Name	Type	Description
	<pre>const double[] for real data in double precision flavors,</pre> <pre>const MKL_Complex8[] for complex data in single precision flavors,</pre> <pre>const MKL_Complex16[] for complex data in double precision flavors</pre>	
xstride	<pre>FORTRAN 77: INTEGER</pre> <pre>Fortran 90: INTEGER</pre> <pre>C: const MKL_INT</pre>	Stride for input data sequence in the arrayx.

Output Parameters

Name	Type	Description
task	<pre>FORTRAN 77: INTEGER*4 task(2) for vslsconvnewtaskx1d, vsldconvnewtaskx1d, vslcconvnewtaskx1d, vslzconvnewtaskx1d</pre> <pre>INTEGER*4 task(2) for vslscornewtaskx1d, vsldcornewtaskx1d, vslccornewtaskx1d, vslzcornewtaskx1d</pre> <pre>Fortran 90: TYPE(VSL_CONV_TASK) for vslsconvnewtaskx1d, vsldconvnewtaskx1d, vslcconvnewtaskx1d, vslzconvnewtaskx1d</pre> <pre>TYPE(VSL_CORR_TASK) for vslscornewtaskx1d, vsldcornewtaskx1d, vslccornewtaskx1d, vslzcornewtaskx1d</pre> <pre>C: VSLConvTaskPtr* for vslsConvNewTaskX1D, vsldConvNewTaskX1D, vslcConvNewTaskX1D, vslzConvNewTaskX1D</pre>	Pointer to the task descriptor if created successfully or <code>NULL</code> pointer otherwise.

Name	Type	Description
	VSLCorrTaskPtr* for vslsCorrNewTaskX1D, vsldCorrNewTaskX1D, vslcCorrNewTaskX1D, vslzCorrNewTaskX1D	
status	FORTRAN 77: INTEGER Fortran 90: INTEGER C: int	Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.

Description

Each `vslConvNewTaskX1D/vslCorrNewTaskX1D` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see [Table "Convolution and Correlation Task Parameters"](#)).

These routines represent a special one-dimensional version of the so called X-form of the constructor. This assumes that the value of the parameter `dims` is 1 and that in addition to creating the task descriptor, constructor routines assign particular data to the first operand vector in array `x` used in convolution or correlation operation. The task descriptor created by the `vslConvNewTaskX1D/vslCorrNewTaskX1D` constructor keeps the pointer to the array `x` all the time, that is, until the task object is deleted by one of the destructor routines (see [vslConvDeleteTask/vslCorrDeleteTask](#)).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array `x` against different vectors in array `y`. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters `xshape`, `yshape`, and `zshape` are equal to the number of elements read from the arrays `x` and `y` or stored to the array `z`. You explicitly assign the shape parameters when calling the constructor.

The **stride parameters** `xstride` specifies the physical location of the input data in the array `x` and is an interval between locations of consecutive elements of the array. For example, if the value of the parameter `xstride` is `s`, then only every `s`th element of the array `x` will be used to form the input sequence. The stride value must be positive or negative but not zero.

Task Editors

Task editors in convolution and correlation API of Intel MKL are routines intended for setting up or changing the following task parameters (see [Table "Convolution and Correlation Task Parameters"](#)):

- `mode`
- `internal_precision`
- `start`
- `decimation`

For setting up or changing each of the above parameters, a separate routine exists.

NOTE

Fields of the task descriptor structure are accessible only through the set of task editor routines provided with the software.

The work data computed during the last commitment process may become invalid with respect to new parameter settings. That is why after applying any of the editor routines to change the task descriptor settings, the task loses its commitment status and goes through the full commitment process again during the next execution or copy operation. For more information on task commitment, see the [Introduction to Convolution and Correlation](#).

Table "Task Editors" lists available task editors.

Task Editors

Routine	Description
<code>vslConvSetMode/vslCorrSetMode</code>	Changes the value of the parameter <i>mode</i> for the operation of convolution or correlation.
<code>vslConvSetInternalPrecision/ vslCorrSetInternalPrecision</code>	Changes the value of the parameter <i>internal_precision</i> for the operation of convolution or correlation.
<code>vslConvSetStart/vslCorrSetStart</code>	Sets the value of the parameter <i>start</i> for the operation of convolution or correlation.
<code>vslConvSetDecimation/ vslCorrSetDecimation</code>	Sets the value of the parameter <i>decimation</i> for the operation of convolution or correlation.

NOTE

You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

vslConvSetMode/vslCorrSetMode

Changes the value of the parameter mode in the convolution or correlation task descriptor.

Syntax

Fortran:

```
status = vslconvsetmode(task, newmode)
status = vslcorrsetmode(task, newmode)
```

C:

```
status = vslConvSetMode(task, newmode);
status = vslCorrSetMode(task, newmode);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>FORTTRAN 77:</code> <code>INTEGER*4 task(2) for vslconvsetmode</code>	Pointer to the task descriptor.

Name	Type	Description
	INTEGER*4 task(2) for vslcorrsetmode	
	Fortran 90: TYPE(VSL_CONV_TASK) for vslconvsetmode	
	TYPE(VSL_CORR_TASK) for vslcorrsetmode	
	C: VSLConvTaskPtr for vslConvSetMode	
	VSLCorrTaskPtr for vslCorrSetMode	
newmode	FORTRAN 77: INTEGER Fortran 90: INTEGER C: const MKL_INT	New value of the parameter mode.

Output Parameters

Name	Type	Description
status	FORTRAN 77: INTEGER Fortran 90: INTEGER C: int	Current status of the task.

Description

This function is declared in `mkl_vsl.f77` for the FORTRAN 77 interface, in `mkl_vsl.f90` for the Fortran 90 interface, and in `mkl_vsl_functions.h` for the C interface.

The function routine changes the value of the parameter `mode` for the operation of convolution or correlation. This parameter defines whether the computation should be done via Fourier transforms of the input/output data or using a direct algorithm. Initial value for `mode` is assigned by a task constructor.

Predefined values for the `mode` parameter are as follows:

Values of `mode` parameter

Value	Purpose
<code>VSL_CONV_MODE_FFT</code>	Compute convolution by using fast Fourier transform.
<code>VSL_CORR_MODE_FFT</code>	Compute correlation by using fast Fourier transform.
<code>VSL_CONV_MODE_DIRECT</code>	Compute convolution directly.
<code>VSL_CORR_MODE_DIRECT</code>	Compute correlation directly.
<code>VSL_CONV_MODE_AUTO</code>	Automatically choose direct or Fourier mode for convolution.
<code>VSL_CORR_MODE_AUTO</code>	Automatically choose direct or Fourier mode for correlation.

vslConvSetInternalPrecision/vslCorrSetInternalPrecision

Changes the value of the parameter `internal_precision` in the convolution or correlation task descriptor.

Syntax

Fortran:

```
status = vslconvsetinternalprecision(task, precision)
status = vslcorrsetinternalprecision(task, precision)
```

C:

```
status = vslConvSetInternalPrecision(task, precision);
status = vslCorrSetInternalPrecision(task, precision);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	FORTRAN 77: INTEGER*4 task(2) for vslconvsetinternalprecision INTEGER*4 task(2) for vslcorrsetinternalprecision Fortran 90: TYPE(VSL_CONV_TASK) for vslconvsetinternalprecision TYPE(VSL_CORR_TASK) for vslcorrsetinternalprecision C: VSLConvTaskPtr for vslConvSetInternalPrecision VSLCorrTaskPtr for vslCorrSetInternalPrecision	Pointer to the task descriptor.
<i>precision</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER C: const MKL_INT	New value of the parameter <code>internal_precision</code> .

Output Parameters

Name	Type	Description
<i>status</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER C: int	Current status of the task.

Description

The `vslConvSetInternalPrecision/vslCorrSetInternalPrecision` routine changes the value of the parameter `internal_precision` for the operation of convolution or correlation. This parameter defines whether the internal computations of the convolution or correlation result should be done in single or double precision. Initial value for `internal_precision` is assigned by a task constructor and set to either "single" or "double" according to the particular flavor of the constructor used.

Changing the `internal_precision` can be useful if the default setting of this parameter was "single" but you want to calculate the result with double precision even if input and output data are represented in single precision.

Predefined values for the `internal_precision` input parameter are as follows:

Values of `internal_precision` Parameter

Value	Purpose
<code>VSL_CONV_PRECISION_SINGLE</code>	Compute convolution with single precision.
<code>VSL_CORR_PRECISION_SINGLE</code>	Compute correlation with single precision.
<code>VSL_CONV_PRECISION_DOUBLE</code>	Compute convolution with double precision.
<code>VSL_CORR_PRECISION_DOUBLE</code>	Compute correlation with double precision.

`vslConvSetStart/vslCorrSetStart`

Changes the value of the parameter `start` in the convolution or correlation task descriptor.

Syntax

Fortran:

```
status = vslconvsetstart(task, start)
status = vslcorrsetstart(task, start)
```

C:

```
status = vslConvSetStart(task, start);
status = vslCorrSetStart(task, start);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	<code>FORTRAN 77:</code> <code>INTEGER*4 task(2) for</code> <code>vslconvsetstart</code> <code>INTEGER*4 task(2) for</code> <code>vslcorrsetstart</code>	Pointer to the task descriptor.

Name	Type	Description
	Fortran 90: TYPE(VSL_CONV_TASK) for vslconvsetstart	
	TYPE(VSL_CORR_TASK) for vslcorrsetstart	
	C: VSLConvTaskPtr for vslConvSetStart	
	VSLCorrTaskPtr for vslCorrSetStart	
<i>start</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION (*) C: const int[]	New value of the parameter <i>start</i> .

Output Parameters

Name	Type	Description
<i>status</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER C: int	Current status of the task.

Description

The `vslConvSetStart/vslCorrSetStart` routine sets the value of the parameter *start* for the operation of convolution or correlation. In a one-dimensional case, this parameter points to the first element in the mathematical result that should be stored in the output array. In a multidimensional case, *start* is an array of indices and its length is equal to the number of dimensions specified by the parameter *dims*. For more information about the definition and effect of this parameter, see [Data Allocation](#).

During the initial task descriptor construction, the default value for *start* is undefined and this parameter is not used. Therefore the only way to set and use the *start* parameter is via assigning it some value by one of the `vslConvSetStart/vslCorrSetStart` routines.

vslConvSetDecimation/vslCorrSetDecimation

Changes the value of the parameter decimation in the convolution or correlation task descriptor.

Syntax

Fortran:

```
status = vslconvsetdecimation(task, decimation)
status = vslcorrsetdecimation(task, decimation)
```

C:

```
status = vslConvSetDecimation(task, decimation);
status = vslCorrSetDecimation(task, decimation);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<code>task</code>	FORTRAN 77: INTEGER*4 task(2) for vslconvsetdecimation INTEGER*4 task(2) for vslcorrsetdecimation Fortran 90: TYPE(VSL_CONV_TASK) for vslconvsetdecimation TYPE(VSL_CORR_TASK) for vslcorrsetdecimation C: VSLConvTaskPtr for vslConvSetDecimation VSLCorrTaskPtr for vslCorrSetDecimation	Pointer to the task descriptor.
<code>decimation</code>	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION (*) C: const int[]	New value of the parameter <code>decimation</code> .

Output Parameters

Name	Type	Description
<code>status</code>	FORTRAN 77: INTEGER Fortran 90: INTEGER C: int	Current status of the task.

Description

The routine sets the value of the parameter `decimation` for the operation of convolution or correlation. This parameter determines how to thin out the mathematical result of convolution or correlation before writing it into the output data array. For example, in a one-dimensional case, if `decimation` = $d > 1$, only every d -th element of the mathematical result is written to the output array z . In a multidimensional case, `decimation` is an array of indices and its length is equal to the number of dimensions specified by the parameter `dims`. For more information about the definition and effect of this parameter, see [Data Allocation](#).

During the initial task descriptor construction, the default value for `decimation` is undefined and this parameter is not used. Therefore the only way to set and use the `decimation` parameter is via assigning it some value by one of the `vslSetDecimation` routines.

Task Execution Routines

Task execution routines compute convolution or correlation results based on parameters held by the task descriptor and on the user data supplied for input vectors.

After you create and adjust a task, you can execute it multiple times by applying to different input/output data of the same type, precision, and shape.

Intel MKL provides the following forms of convolution/correlation execution routines:

- **General form** executors that use the task descriptor created by the general form constructor and expect to get two source data arrays x and y on input
- **X-form** executors that use the task descriptor created by the X-form constructor and expect to get only one source data array y on input because the first array x has been already specified on the construction stage

When the task is executed for the first time, the execution routine includes a task commitment operation, which involves two basic steps: parameters consistency check and preparation of auxiliary data (for example, this might be the calculation of Fourier transform for input data).

Each execution routine has an associated one-dimensional version that provides algorithmic and computational benefits.

NOTE

You can use the `NULL` task pointer in calls to execution routines. In this case, the routine is terminated and no system crash occurs.

If the task is executed successfully, the execution routine returns the zero status code. If an error is detected, the execution routine returns an error code which signals that a specific error has occurred. In particular, an error status code is returned in the following cases:

- if the task pointer is `NULL`
- if the task descriptor is corrupted
- if calculation has failed for some other reason.

NOTE

Intel® MKL does not control floating-point errors, like overflow or gradual underflow, or operations with NaNs, etc.

If an error occurs, the task descriptor stores the error code.

The table below lists all task execution routines.

Task Execution Routines

Routine	Description
vslConvExec/vslCorrExec	Computes convolution or correlation for a multidimensional case.
vslConvExec1D/vslCorrExec1D	Computes convolution or correlation for a one-dimensional case.
vslConvExecX/vslCorrExecX	Computes convolution or correlation as X-form for a multidimensional case.
vslConvExecX1D/vslCorrExecX1D	Computes convolution or correlation as X-form for a one-dimensional case.

[vslConvExec/vslCorrExec](#)

Computes convolution or correlation for multidimensional case.

Syntax

Fortran:

```
status = vslsconvexec(task, x, xstride, y, ystride, z, zstride)
status = vsldconvexec(task, x, xstride, y, ystride, z, zstride)
status = vslcconvexec(task, x, xstride, y, ystride, z, zstride)
status = vslzconvexec(task, x, xstride, y, ystride, z, zstride)
status = vslscorrexec(task, x, xstride, y, ystride, z, zstride)
status = vsldcorrexec(task, x, xstride, y, ystride, z, zstride)
status = vslccorrexec(task, x, xstride, y, ystride, z, zstride)
status = vslzcorrexec(task, x, xstride, y, ystride, z, zstride)
```

C:

```
status = vslsConvExec(task, x, xstride, y, ystride, z, zstride);
status = vsldConvExec(task, x, xstride, y, ystride, z, zstride);
status = vslcConvExec(task, x, xstride, y, ystride, z, zstride);
status = vslzConvExec(task, x, xstride, y, ystride, z, zstride);
status = vslsCorrExec(task, x, xstride, y, ystride, z, zstride);
status = vsldCorrExec(task, x, xstride, y, ystride, z, zstride);
status = vslcCorrExec(task, x, xstride, y, ystride, z, zstride);
status = vslzCorrExec(task, x, xstride, y, ystride, z, zstride);
```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
task	FORTRAN 77: INTEGER*4 task(2) for vslsconvexec, vsldconvexec, vslcconvexec, vslzconvexec INTEGER*4 task(2) for vslscorrexec, vsldcorrexec, vslccorrexec, vslzcorrexec Fortran 90: TYPE(VSL_CONV_TASK) for vslsconvexec, vsldconvexec, vslcconvexec, vslzconvexec TYPE(VSL_CORR_TASK) for vslscorrexec, vsldcorrexec, vslccorrexec, vslzcorrexec	Pointer to the task descriptor

Name	Type	Description
	<p>C: VSLConvTaskPtr for vslsConvExec, vsldConvExec, vslcConvExec, vslzConvExec</p> <p>VSLCorrTaskPtr for vslsCorrExec, vsldCorrExec, vslcCorrExec, vslzCorrExec</p>	
x, y	<p>FORTRAN 77: REAL*4 for vslsconvexec and vslscorrexec,</p> <p>REAL*8 for vsldconvexec and vsldcorrexec,</p> <p>COMPLEX*8 for vslcconvexec and vslccorrexec,</p> <p>COMPLEX*16 for vslzconvexec and vslzcorrexec</p> <p>Fortran 90:</p> <p>REAL(KIND=4), DIMENSION(*) for vslsconvexec and vslscorrexec,</p> <p>REAL(KIND=8), DIMENSION(*) for vsldconvexec and vsldcorrexec,</p> <p>COMPLEX(KIND=4), DIMENSION(*) for vslcconvexec and vslccorrexec,</p> <p>COMPLEX(KIND=8), DIMENSION(*) for vslzconvexec and vslzcorrexec</p> <p>C: const float[] for vslsConvExec and vslsCorrExec,</p> <p>const double[] for vsldConvExec and vsldCorrExec,</p> <p>const MKL_Complex8[] for vslcConvExec and vslcCorrExec,</p> <p>const MKL_Complex16[] for vslzConvExec and vslzCorrExec</p>	Pointers to arrays containing input data. See Data Allocation for more information.
xstride, ystride, zstride	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90:</p> <p>INTEGER, DIMENSION (*)</p>	Strides for input and output data. For more information, see stride parameters .

Name	Type	Description
------	------	-------------

C: const int[]

Output Parameters

Name	Type	Description
<i>z</i>	<p>FORTRAN 77: REAL*4 for vslsconvexec and vslscorrexec,</p> <p>REAL*8 for vsldconvexec and vsldcorrexec,</p> <p>COMPLEX*8 for vslcconvexec and vslccorrexec,</p> <p>COMPLEX*16 for vslzconvexec and vslzcorrexec</p> <p>Fortran 90:</p> <p>REAL(KIND=4), DIMENSION(*) for vslsconvexec and vslscorrexec,</p> <p>REAL(KIND=8), DIMENSION(*) for vsldconvexec and vsldcorrexec,</p> <p>COMPLEX(KIND=4), DIMENSION(*) for vslcconvexec and vslccorrexec,</p> <p>COMPLEX(KIND=8), DIMENSION(*) for vslzconvexec and vslzcorrexec</p> <p>C: const float[] for vslsConvExec and vslsCorrExec,</p> <p>const double[] for vsldConvExec and vsldCorrExec,</p> <p>const MKL_Complex8[] for vslcConvExec and vslcCorrExec,</p> <p>const MKL_Complex16[] for vslzConvExec and vslzCorrExec</p>	Pointer to the array that stores output data. See Data Allocation for more information.
<i>status</i>	<p>FORTRAN 77: INTEGER</p> <p>Fortran 90: INTEGER</p> <p>C: int</p>	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExec/vslCorrExec` routines computes convolution or correlation of the data provided by the arrays `x` and `y` and then stores the results in the array `z`. Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTask/vslCorrNewTask` constructor and pointed to by `task`. If `task` is `NULL`, no operation is done.

The stride parameters `xstride`, `ystride`, and `zstride` specify the physical location of the input and output data in the arrays `x`, `y`, and `z`, respectively. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter `zstride` is `s`, then only every s^{th} element of the array `z` will be used to store the output data. The stride value must be positive or negative but not zero.

vslConvExec1D/vslCorrExec1D

Computes convolution or correlation for one-dimensional case.

Syntax

Fortran:

```
status = vslsconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vsldconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslcconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslzconvexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslscorrexec1d(task, x, xstride, y, ystride, z, zstride)
status = vsldcorrexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslccorrexec1d(task, x, xstride, y, ystride, z, zstride)
status = vslzcorrexec1d(task, x, xstride, y, ystride, z, zstride)
```

C:

```
status = vslsConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vsldConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslcConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslzConvExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslsCorrExec1D(task, x, xstride, y, ystride, z, zstride);
status = vsldCorrExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslcCorrExec1D(task, x, xstride, y, ystride, z, zstride);
status = vslzCorrExec1D(task, x, xstride, y, ystride, z, zstride);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	<p>FORTRAN 77:</p> <pre>INTEGER*4 task(2) for vslsconvexec1d, vsldconvexec1d, vslcconvexec1d, vslzconvexec1d</pre> <p>INTEGER*4 task(2) for vslscorrexec1d, vsldcorrexec1d, vslccorrexec1d, vslzcorrexec1d</p> <p>Fortran 90:</p> <pre>TYPE(VSL_CONV_TASK) for vslsconvexec1d, vsldconvexec1d, vslcconvexec1d, vslzconvexec1d</pre> <p>TYPE(VSL_CORR_TASK) for vslscorrexec1d, vsldcorrexec1d, vslccorrexec1d, vslzcorrexec1d</p> <p>C: VSLConvTaskPtr for vslsConvExec1D, vsldConvExec1D, vslcConvExec1D, vslzConvExec1D</p> <p>VSLCorrTaskPtr for vslsCorrExec1D, vsldCorrExec1D, vslcCorrExec1D, vslzCorrExec1D</p>	Pointer to the task descriptor.
<i>x, y</i>	<p>FORTRAN 77: REAL*4 for vslsconvexec1d and vslscorrexec1d,</p> <p>REAL*8 for vsldconvexec1d and vsldcorrexec1d,</p> <p>COMPLEX*8 for vslcconvexec1d and vslccorrexec1d,</p> <p>COMPLEX*16 for vslzconvexec1d and vslzcorrexec1d</p>	Pointers to arrays containing input data. See Data Allocation for more information.

Name	Type	Description
	<pre> Fortran 90: REAL(KIND=4), DIMENSION(*) for vslsconvexec1d and vslscorrexec1d, REAL(KIND=8), DIMENSION(*) for vsldconvexec1d and vsldcorrexec1d, COMPLEX(KIND=4), DIMENSION (*) for vslcconvexec1d and vslccorrexec1d, COMPLEX(KIND=8), DIMENSION (*) for vslzconvexec1d and vslzcorrexec1d C: const float[] for vslsConvExec1D and vslsCorrExec1D, const double[] for vsldConvExec1D and vsldCorrExec1D, const MKL_Complex8[] for vslcConvExec1D and vslcCorrExec1D, const MKL_Complex16[] for vslzConvExec1D and vslzCorrExec1D </pre>	
xstride, ystride, zstride	FORTRAN 77: INTEGER Fortran 90: INTEGER C: const MKL_INT	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
z	FORTRAN 77: REAL*4 for vslsconvexec1d and vslscorrexec1d, REAL*8 for vsldconvexec1d and vsldcorrexec1d, COMPLEX*8 for vslcconvexec1d and vslccorrexec1d, COMPLEX*16 for vslzconvexec1d and vslzcorrexec1d	Pointer to the array that stores output data. See Data Allocation for more information.

Name	Type	Description
	Fortran 90:	
	REAL(KIND=4), DIMENSION(*) for vslsconvexec1d and vslscorrexec1d,	
	REAL(KIND=8), DIMENSION(*) for vsldconvexec1d and vsldcorrexec1d,	
	COMPLEX(KIND=4), DIMENSION (*) for vslcconvexec1d and vslccorrexec1d,	
	COMPLEX(KIND=8), DIMENSION (*) for vslzconvexec1d and vslzcorrexec1d	
	C: const float[] for vslsConvExec1D and vslsCorrExec1D, const double[] for vsldConvExec1D and vsldCorrExec1D, const MKL_Complex8[] for vslcConvExec1D and vslcCorrExec1D, const MKL_Complex16[] for vslzConvExec1D and vslzCorrExec1D	
status	FORTRAN 77: INTEGER Fortran 90: INTEGER C: int	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExec1D/vslCorrExec1D` routines computes convolution or correlation of the data provided by the arrays `x` and `y` and then stores the results in the array `z`. These routines represent a special one-dimensional version of the operation, assuming that the value of the parameter `dims` is 1. Using this version of execution routines can help speed up performance in case of one-dimensional data.

Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTask1D/vslCorrNewTask1D` constructor and pointed to by `task`. If `task` is NULL, no operation is done.

vslConvExecX/vslCorrExecX

Computes convolution or correlation for multidimensional case with the fixed first operand vector.

Syntax

Fortran:

```

status = vslsconvexecx(task, y, ystride, z, zstride)
status = vslldconvexecx(task, y, ystride, z, zstride)
status = vslcconvexecx(task, y, ystride, z, zstride)
status = vslzconvexecx(task, y, ystride, z, zstride)
status = vsls correxecx(task, y, ystride, z, zstride)
status = vslld correxecx(task, y, ystride, z, zstride)
status = vslccorrexe cx(task, y, ystride, z, zstride)
status = vslz correxe cx(task, y, ystride, z, zstride)

```

C:

```

status = vslsConvExecX(task, y, ystride, z, zstride);
status = vslldConvExecX(task, y, ystride, z, zstride);
status = vslcConvExecX(task, y, ystride, z, zstride);
status = vslzConvExecX(task, y, ystride, z, zstride);
status = vslsCorrExecX(task, y, ystride, z, zstride);
status = vslcCorrExecX(task, y, ystride, z, zstride);
status = vslzCorrExecX(task, y, ystride, z, zstride);
status = vslldCorrExecX(task, y, ystride, z, zstride);

```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
task	FORTRAN 77: INTEGER*4 task(2) for vslsconvexecx, vslldconvexecx, vslcconvexecx, vslzconvexecx INTEGER*4 task(2) for vsls correxecx, vslld correxecx, vslccorrexe cx, vslz correxe cx Fortran 90: TYPE(VSL_CONV_TASK) for vslsconvexecx, vslldconvexecx, vslcconvexecx, vslzconvexecx	Pointer to the task descriptor.

Name	Type	Description
	<pre>TYPE(VSL_CORR_TASK) for vslscorrexecx, vsldcorrexecx, vslccorrexecx, vslzcorrexecx</pre>	
	<pre>C: VSLConvTaskPtr for vslsConvExecX, vsldConvExecX, vslcConvExecX, vslzConvExecX</pre>	
	<pre>VSLCorrTaskPtr for vslsCorrExecX, vsldCorrExecX, vslcCorrExecX, vslzCorrExecX</pre>	
x ,y	<pre>FORTRAN 77: REAL*4 for vslsconvexecx and vslscorrexecx,</pre> <pre>REAL*8 for vsldconvexecx and vsldcorrexecx,</pre> <pre>COMPLEX*8 for vslccconvexecx and vslccorrexecx,</pre> <pre>COMPLEX*16 for vslzconvexecx and vslzcorrexecx</pre> <pre>Fortran 90:</pre> <pre>REAL(KIND=4), DIMENSION(*) for vslsconvexecx and vslscorrexecx,</pre> <pre>REAL(KIND=8), DIMENSION(*) for vsldconvexecx and vsldcorrexecx,</pre> <pre>COMPLEX(KIND=4), DIMENSION (*) for vslccconvexecx and vslccorrexecx,</pre> <pre>COMPLEX(KIND=8), DIMENSION (*) for vslzconvexecx and vslzcorrexecx</pre> <pre>C: const float[] for vslsConvExecX and vslsCorrExecX,</pre> <pre>const double[] for vsldConvExecX and vsldCorrExecX,</pre>	Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.

Name	Type	Description
	<pre>const MKL_Complex8[] for vslcConvExecX and vslcCorrExecX,</pre> <pre>const MKL_Complex16[] for vslzConvExecX and vslzCorrExecX</pre>	
<i>ystride</i> , <i>zstride</i>	FORTRAN 77: INTEGER Fortran 90: INTEGER, DIMENSION (*) C: const int[]	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
<i>z</i>	FORTRAN 77: REAL*4 for vslsconvexecx and vslscorrexecx, REAL*8 for vsldconvexecx and vsldcorrexecx, COMPLEX*8 for vslcconvexecx and vslccorrexecx, COMPLEX*16 for vslzconvexecx and vslzcorrexecx Fortran 90: REAL(KIND=4), DIMENSION(*) for vslsconvexecx and vslscorrexecx, REAL(KIND=8), DIMENSION(*) for vsldconvexecx and vsldcorrexecx, COMPLEX(KIND=4), DIMENSION (*) for vslcconvexecx and vslccorrexecx, COMPLEX(KIND=8), DIMENSION (*) for vslzconvexecx and vslzcorrexecx C: const float[] for vslsConvExecX and vslsCorrExecX, const double[] for vsldConvExecX and vsldCorrExecX,	Pointer to the array that stores output data. See Data Allocation for more information.

Name	Type	Description
	const MKL_Complex8[] for vslcConvExecX and vslcCorrExecX, const MKL_Complex16[] for vslzConvExecX and vslzCorrExecX	
status	FORTRAN 77: INTEGER Fortran 90: INTEGER C: int	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the vslConvExecX/vslCorrExecX routines computes convolution or correlation of the data provided by the arrays x and y and then stores the results in the array z . These routines represent a special version of the operation, which assumes that the first operand vector was set on the task construction stage and the task object keeps the pointer to the array x .

Parameters of the operation are read from the task descriptor created previously by a corresponding [vslConvNewTaskX/vslCorrNewTaskX](#) constructor and pointed to by $task$. If $task$ is NULL, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple convolutions or correlations with the same data vector in array x against different vectors in array y . This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

vslConvExecX1D/vslCorrExecX1D

Computes convolution or correlation for one-dimensional case with the fixed first operand vector.

Syntax

Fortran:

```
status = vslsconvexecx1d(task, y, ystride, z, zstride)
status = vsldconvexecx1d(task, y, ystride, z, zstride)
status = vslcconvexecx1d(task, y, ystride, z, zstride)
status = vslzconvexecx1d(task, y, ystride, z, zstride)
status = vslscorrexecx1d(task, y, ystride, z, zstride)
status = vslldcorrexecx1d(task, y, ystride, z, zstride)
status = vslccorrexecx1d(task, y, ystride, z, zstride)
status = vslzcorrexecx1d(task, y, ystride, z, zstride)
```

C:

```
status = vslsConvExecX1D(task, y, ystride, z, zstride);
status = vsldConvExecX1D(task, y, ystride, z, zstride);
status = vslcConvExecX1D(task, y, ystride, z, zstride);
status = vslzConvExecX1D(task, y, ystride, z, zstride);
```

```

status = vslsCorrExecX1D(task, y, ystride, z, zstride);
status = vslcCorrExecX1D(task, y, ystride, z, zstride);
status = vslzCorrExecX1D(task, y, ystride, z, zstride);
status = vsldCorrExecX1D(task, y, ystride, z, zstride);

```

Include Files

- Fortran: mkl.fi
- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
task	FORTRAN 77: INTEGER*4 task(2) for vslsconvexec1d, vsldconvexec1d, vslcconvexec1d, vslzconvexec1d INTEGER*4 task(2) for vslscorrexec1d, vsldcorrexec1d, vslccorrexec1d, vslzcorrexec1d Fortran 90: TYPE(VSL_CONV_TASK) for vslsconvexec1d, vsldconvexec1d, vslcconvexec1d, vslzconvexec1d TYPE(VSL_CORR_TASK) for vslscorrexec1d, vsldcorrexec1d, vslccorrexec1d, vslzcorrexec1d C: VSLConvTaskPtr for vslsConvExec1D, vsldConvExec1D, vslcConvExec1D, vslzConvExec1D VSLCorrTaskPtr for vslsCorrExec1D, vsldCorrExec1D, vslcCorrExec1D, vslzCorrExec1D	Pointer to the task descriptor.
x , y	FORTRAN 77: REAL*4 for vslsconvexec1d and vslscorrexec1d,	Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.

Name	Type	Description
	<pre>REAL*8 for vslconvexecx1d and vslcorrexecx1d, COMPLEX*8 for vslcconvexecx1d and vslccorrexecx1d, COMPLEX*16 for vslzconvexecx1d and vslzcorrexecx1d Fortran 90: REAL(KIND=4), DIMENSION(*) for vslsconvexecx1d and vslscorrexecx1d, REAL(KIND=8), DIMENSION(*) for vsldconvexecx1d and vsldcorrexecx1d, COMPLEX(KIND=4), DIMENSION (*) for vslcconvexecx1d and vslccorrexecx1d, COMPLEX(KIND=8), DIMENSION (*) for vslzconvexecx1d and vslzcorrexecx1d C: const float[] for vslsConvExecX1D and vslsCorrExecX1D, const double[] for vsldConvExecX1D and vsldCorrExecX1D, const MKL_Complex8[] for vslcConvExecX1D and vslcCorrExecX1D, const MKL_Complex16[] for vslzConvExecX1D and vslzCorrExecX1D</pre>	
ystride, zstride	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER C: const MKL_INT</pre>	Strides for input and output data. For more information, see stride parameters .

Output Parameters

Name	Type	Description
z	<pre>FORTRAN 77: REAL*4 for vslsconvexecx1d and vslscorrexecx1d,</pre>	Pointer to the array that stores output data. See Data Allocation for more information.

Name	Type	Description
	<pre>REAL*8 for vsldconvexecx1d and vsldcorreexecx1d, COMPLEX*8 for vslcconvexecx1d and vslccorreexecx1d, COMPLEX*16 for vslzconvexecx1d and vslzcorreexecx1d Fortran 90: REAL(KIND=4), DIMENSION(*) for vslsconvexecx1d and vsllcorreexecx1d, REAL(KIND=8), DIMENSION(*) for vsldconvexecx1d and vsldcorreexecx1d, COMPLEX(KIND=4), DIMENSION (*) for vslcconvexecx1d and vslccorreexecx1d, COMPLEX(KIND=8), DIMENSION (*) for vslzconvexecx1d and vslzcorreexecx1d C: const float[] for vsllsConvExecX1D and vsllsCorrExecX1D, const double[] for vsldConvExecX1D and vsldCorrExecX1D, const MKL_Complex8[] for vslcConvExecX1D and vslcCorrExecX1D, const MKL_Complex16[] for vslzConvExecX1D and vslzCorrExecX1D</pre>	
status	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER C: int</pre>	Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

Description

Each of the `vslConvExecX1D/vslCorrExecX1D` routines computes convolution or correlation of one-dimensional (assuming that `dims = 1`) data provided by the arrays `x` and `y` and then stores the results in the array `z`. These routines represent a special version of the operation, which expects that the first operand vector was set on the task construction stage.

Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTaskX1D/vslCorrNewTaskX1D` constructor and pointed to by `task`. If `task` is NULL, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple one-dimensional convolutions or correlations with the same data vector in array `x` against different vectors in array `y`. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Task Destroyors

Task destructors are routines designed for deleting task objects and deallocating memory.

vslConvDeleteTask/vslCorrDeleteTask

Destroys the task object and frees the memory.

Syntax

Fortran:

```
errcode = vslconvdeletetask(task)
errcode = vslcorrdeletetask(task)
```

C:

```
errcode = vslConvDeleteTask(task);
errcode = vslCorrDeleteTask(task);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	FORTRAN 77: INTEGER*4 <code>task(2)</code> for <code>vslconvdeletetask</code> INTEGER*4 <code>task(2)</code> for <code>vslcorrdeletetask</code> Fortran 90: TYPE(VSL_CONV_TASK) for <code>vslconvdeletetask</code> TYPE(VSL_CORR_TASK) for <code>vslcorrdeletetask</code> C: <code>VSLConvTaskPtr*</code> for <code>vslConvDeleteTask</code> <code>VSLCorrTaskPtr*</code> for <code>vslCorrDeleteTask</code>	Pointer to the task descriptor.

Output Parameters

Name	Type	Description
<code>errcode</code>	<code>FORTRAN 77:</code> INTEGER <code>Fortran 90:</code> INTEGER <code>C:</code> int	Contains 0 if the task object is deleted successfully. Contains an error code if an error occurred.

Description

The `vslConvDeleteTask/vslCorrvDeleteTask` routine deletes the task descriptor object and frees any working memory and the memory allocated for the data structure. The task pointer is set to `NULL`.

Note that if the `vslConvDeleteTask/vslCorrvDeleteTask` routine does not delete the task successfully, the routine returns an error code. This error code has no relation to the task status code and does not change it.

NOTE

You can use the `NULL` task pointer in calls to destructor routines. In this case, the routine terminates with no system crash.

Task Copy

The routines are designed for copying convolution and correlation task descriptors.

`vslConvCopyTask/vslCorrCopyTask`

Copies a descriptor for convolution or correlation task.

Syntax

Fortran:

```
status = vslconvcopytask(newtask, srctask)
status = vslcorrcopytask(newtask, srctask)
```

C:

```
status = vslConvCopyTask(newtask, srctask);
status = vslCorrCopyTask(newtask, srctask);
```

Include Files

- Fortran: `mkl.fi`
- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>srctask</code>	<code>FORTRAN 77:</code> INTEGER*4 <code>srctask(2)</code> for <code>vslconvcopytask</code>	Pointer to the source task descriptor.

Name	Type	Description
	<pre>INTEGER*4 srctask(2) for vslcorrcopytask Fortran 90: TYPE(VSL_CONV_TASK) for vslconvcopytask TYPE(VSL_CORR_TASK) for vslcorrcopytask C: const VSLConvTaskPtr for vslConvCopyTask const VSLCorrTaskPtr for vslCorrCopyTask</pre>	

Output Parameters

Name	Type	Description
<i>newtask</i>	<pre>FORTRAN 77: INTEGER*4 srctask(2) for vslconvcopytask INTEGER*4 srctask(2) for vslcorrcopytask Fortran 90: TYPE(VSL_CONV_TASK) for vslconvcopytask TYPE(VSL_CORR_TASK) for vslcorrcopytask C: VSLConvTaskPtr* for vslConvCopyTask VSLCorrTaskPtr* for vslCorrCopyTask</pre>	Pointer to the new task descriptor.
<i>status</i>	<pre>FORTRAN 77: INTEGER Fortran 90: INTEGER C: int</pre>	Current status of the source task.

Description

If a task object *srctask* already exists, you can use an appropriate `vslConvCopyTask/vslCorrCopyTask` routine to make its copy in *newtask*. After the copy operation, both source and new task objects will become committed (see [Introduction to Convolution and Correlation](#) for information about task commitment). If the source task was not previously committed, the commitment operation for this task is implicitly invoked before copying starts. If an error occurs during source task commitment, the task stores the error code in the status field. If an error occurs during copy operation, the routine returns a `NULL` pointer instead of a reference to a new task object.

Usage Examples

This section demonstrates how you can use the Intel MKL routines to perform some common convolution and correlation operations both for single-threaded and multithreaded calculations. The following two sample functions `scond1` and `sconf1` simulate the convolution and correlation functions `SCOND` and `SCONF` found in IBM ESSL* library. The functions assume single-threaded calculations and can be used with C or C++ compilers.

Function `scond1` for Single-Threaded Calculations

```
#include "mkl_vsl.h"

int scond1(
    float h[], int inch,
    float x[], int incx,
    float y[], int incy,
    int nh, int nx, int iy0, int ny)
{
    int status;
    VSLConvTaskPtr task;
    vslsConvNewTask1D(&task,VSL_CONV_MODE_DIRECT,nh,nx,ny);
    vslConvSetStart(task, &iy0);
    status = vslsConvExec1D(task, h,inch, x,incx, y,incy);
    vslConvDeleteTask(&task);
    return status;
}
```

Function `sconf1` for Single-Threaded Calculations

```
#include "mkl_vsl.h"

int sconf1(
    int init,
    float h[], int inc1h,
    float x[], int inc1x, int inc2x,
    float y[], int inc1y, int inc2y,
    int nh, int nx, int m, int iy0, int ny,
    void* aux1, int naux1, void* aux2, int naux2)
{
    int status;
    /* assume that aux1!=0 and naux1 is big enough */
    VSLConvTaskPtr* task = (VSLConvTaskPtr*)aux1;
    if (init != 0)
        /* initialization: */
        status = vslsConvNewTaskX1D(task, VSL_CONV_MODE_FFT,
            nh, nx, ny, h, inc1h);
    if (init == 0) {
        /* calculations: */
        int i;
        vslConvSetStart(*task, &iy0);
        for (i=0; i<m; i++) {
            float* xi = &x[inc2x * i];
            float* yi = &y[inc2y * i];
            /* task is implicitly committed at i==0 */
            status = vslsConvExecX1D(*task, xi, inc1x, yi, inc1y);
        };
    };
    vslConvDeleteTask(task);
    return status;
}
```

Using Multiple Threads

For functions such as `sconf1` described in the previous example, parallel calculations may be more preferable instead of cycling. If $m>1$, you can use multiple threads for invoking the task execution against different data sequences. For such cases, use task copy routines to create m copies of the task object before the calculations stage and then run these copies with different threads. Ensure that you make all necessary parameter adjustments for the task (using [Task Editors](#)) before copying it.

The sample code in this case may look as follows:

```
if (init == 0) {
    int i, status, ss[M];
    VSLConvTaskPtr tasks[M];
    /* assume that M is big enough */
    ...
    vslConvSetStart(*task, &iy0);
    ...
    for (i=0; i<m; i++)
        /* implicit commitment at i==0 */
        vslConvCopyTask(&tasks[i],*task);
    ...
}
```

Then, m threads may be started to execute different copies of the task:

```
...
float* xi = &x[inc2x * i];
float* yi = &y[inc2y * i];
ss[i]=vslsConvExecX1D(tasks[i], xi,inc1x, yi,inc1y);
...
```

And finally, after all threads have finished the calculations, overall status should be collected from all task objects. The following code signals the first error found, if any:

```
...
for (i=0; i<m; i++) {
    status = ss[i];
    if (status != 0) /* 0 means "OK" */
        break;
}
return status;
}; /* end if init==0 */
```

Execution routines modify the task internal state (fields of the task structure). Such modifications may conflict with each other if different threads work with the same task object simultaneously. That is why different threads must use different copies of the task.

Mathematical Notation and Definitions

The following notation is necessary to explain the underlying mathematical definitions used in the text:

$\mathbb{R} = (-\infty, +\infty)$	The set of real numbers.
$\mathbb{Z} = \{0, \pm 1, \pm 2, \dots\}$	The set of integer numbers.
$\mathbb{Z}^N = \mathbb{Z} \times \dots \times \mathbb{Z}$	The set of N -dimensional series of integer numbers.
$p = (p_1, \dots, p_N) \in \mathbb{Z}^N$	N -dimensional series of integers.

$u: \mathbf{Z}^N \rightarrow \mathbf{R}$	Function u with arguments from \mathbf{Z}^N and values from \mathbf{R} .
$u(p) = u(p_1, \dots, p_N)$	The value of the function u for the argument (p_1, \dots, p_N) .
$w = u * v$	Function w is the convolution of the functions u, v .
$w = u \cdot v$	Function w is the correlation of the functions u, v .

Given series $p, q \in \mathbf{Z}^N$:

- series $r = p + q$ is defined as $r^n = p^n + q^n$ for every $n=1, \dots, N$
- series $r = p - q$ is defined as $r^n = p^n - q^n$ for every $n=1, \dots, N$
- series $r = \sup\{p, q\}$ is defined as $r^n = \max\{p^n, q^n\}$ for every $n=1, \dots, N$
- series $r = \inf\{p, q\}$ is defined as $r^n = \min\{p^n, q^n\}$ for every $n=1, \dots, N$
- inequality $p \leq q$ means that $p^n \leq q^n$ for every $n=1, \dots, N$.

A function $u(p)$ is called a finite function if there exist series $P^{\min}, P^{\max} \in \mathbf{Z}^N$ such that:

$$u(p) \neq 0$$

implies

$$P^{\min} \leq p \leq P^{\max}.$$

Operations of convolution and correlation are only defined for finite functions.

Consider functions u, v and series $P^{\min}, P^{\max}, Q^{\min}, Q^{\max} \in \mathbf{Z}^N$ such that:

$$u(p) \neq 0 \text{ implies } P^{\min} \leq p \leq P^{\max}.$$

$$v(q) \neq 0 \text{ implies } Q^{\min} \leq q \leq Q^{\max}.$$

Definitions of linear correlation and linear convolution for functions u and v are given below.

Linear Convolution

If function $w = u * v$ is the convolution of u and v , then:

$$w(r) \neq 0 \text{ implies } R^{\min} \leq r \leq R^{\max},$$

where $R^{\min} = P^{\min} + Q^{\min}$ and $R^{\max} = P^{\max} + Q^{\max}$.

If $R^{\min} \leq r \leq R^{\max}$, then:

$$w(r) = \sum_u(t) \cdot v(r-t) \text{ is the sum for all } t \in \mathbf{Z}^N \text{ such that } T^{\min} \leq t \leq T^{\max},$$

where $T^{\min} = \sup\{P^{\min}, r - Q^{\max}\}$ and $T^{\max} = \inf\{P^{\max}, r - Q^{\min}\}$.

Linear Correlation

If function $w = u \cdot v$ is the correlation of u and v , then:

$$w(r) \neq 0 \text{ implies } R^{\min} \leq r \leq R^{\max},$$

where $R^{\min} = Q^{\min} - P^{\max}$ and $R^{\max} = Q^{\max} - P^{\min}$.

If $R^{\min} \leq r \leq R^{\max}$, then:

$$w(r) = \sum_u(t) \cdot v(r+t) \text{ is the sum for all } t \in \mathbf{Z}^N \text{ such that } T^{\min} \leq t \leq T^{\max},$$

where $T^{\min} = \sup\{P^{\min}, Q^{\min} - r\}$ and $T^{\max} = \inf\{P^{\max}, Q^{\max} - r\}$.

Representation of the functions u, v, w as the input/output data for the Intel MKL convolution and correlation functions is described in the [Data Allocation](#) section below.

Data Allocation

This section explains the relation between:

- mathematical finite functions u, v, w introduced in the section [Mathematical Notation and Definitions](#);
- multi-dimensional input and output data vectors representing the functions u, v, w ;

- arrays u, v, w used to store the input and output data vectors in computer memory

The convolution and correlation routine parameters that determine the allocation of input and output data are the following:

- Data arrays x, y, z
- Shape arrays $xshape, yshape, zshape$
- Strides within arrays $xstride, ystride, zstride$
- Parameters $start, decimation$

Finite Functions and Data Vectors

The finite functions $u(p)$, $v(q)$, and $w(r)$ introduced above are represented as multi-dimensional vectors of input and output data:

```
inputu(i1, ..., idims) for u(p1, ..., pN)
inputv(j1, ..., jdims) for v(q1, ..., qN)
output(k1, ..., kdims) for w(r1, ..., rN).
```

Parameter $dims$ represents the number of dimensions and is equal to N.

The parameters $xshape$, $yshape$, and $zshape$ define the shapes of input/output vectors:

```
inputu(i1, ..., idims) is defined if 1 ≤ in ≤ xshape(n) for every n=1, ..., dims
inputv(j1, ..., jdims) is defined if 1 ≤ jn ≤ yshape(n) for every n=1, ..., dims
output(k1, ..., kdims) is defined if 1 ≤ kn ≤ zshape(n) for every n=1, ..., dims.
```

Relation between the input vectors and the functions u and v is defined by the following formulas:

```
inputu(i1, ..., idims) = u(p1, ..., pN), where pn = Pnmin + (in-1) for every n
inputv(j1, ..., jdims) = v(q1, ..., qN), where qn = Qnmin + (jn-1) for every n.
```

The relation between the output vector and the function $w(r)$ is similar (but only in the case when parameters $start$ and $decimation$ are not defined):

```
output(k1, ..., kdims) = w(r1, ..., rN), where rn = Rnmin + (kn-1) for every n.
```

If the parameter $start$ is defined, it must belong to the interval $R_n^{min} ≤ start(n) ≤ R_n^{max}$. If defined, the $start$ parameter replaces R^{min} in the formula:

```
output(k1, ..., kdims) = w(r1, ..., rN), where rn = start(n) + (kn-1)
```

If the parameter $decimation$ is defined, it changes the relation according to the following formula:

```
output(k1, ..., kdims) = w(r1, ..., rN), where rn = Rnmin + (kn-1) * decimation(n)
```

If both parameters $start$ and $decimation$ are defined, the formula is as follows:

```
output(k1, ..., kdims) = w(r1, ..., rN), where rn = start(n) + (kn-1) * decimation(n)
```

The convolution and correlation software checks the values of $zshape$, $start$, and $decimation$ during task commitment. If r_n exceeds R_n^{max} for some $k_n, n=1, ..., dims$, an error is raised.

Allocation of Data Vectors

Both parameter arrays x and y contain input data vectors in memory, while array z is intended for storing output data vector. To access the memory, the convolution and correlation software uses only pointers to these arrays and ignores the array shapes.

For parameters x , y , and z , you can provide one-dimensional arrays with the requirement that actual length of these arrays be sufficient to store the data vectors.

The allocation of the input and output data inside the arrays x , y , and z is described below assuming that the arrays are one-dimensional. Given multi-dimensional indices $i, j, k \in \mathbf{Z}^N$, one-dimensional indices $e, f, g \in \mathbf{Z}$ are defined such that:

```
inputu( $i_1, \dots, i_{\text{dims}}$ ) is allocated at  $x(e)$   

inputv( $j_1, \dots, j_{\text{dims}}$ ) is allocated at  $y(f)$   

output( $k_1, \dots, k_{\text{dims}}$ ) is allocated at  $z(g)$ .
```

The indices e , f , and g are defined as follows:

$$\begin{aligned} e &= 1 + \sum x\text{stride}(n) \cdot dx(n) \quad (\text{the sum is for all } n=1, \dots, \text{dims}) \\ f &= 1 + \sum y\text{stride}(n) \cdot dy(n) \quad (\text{the sum is for all } n=1, \dots, \text{dims}) \\ g &= 1 + \sum z\text{stride}(n) \cdot dz(n) \quad (\text{the sum is for all } n=1, \dots, \text{dims}) \end{aligned}$$

The distances $dx(n)$, $dy(n)$, and $dz(n)$ depend on the signum of the stride:

$$\begin{aligned} dx(n) &= i_n - 1 \text{ if } x\text{stride}(n) > 0, \text{ or } dx(n) = i_n - x\text{shape}(n) \text{ if } x\text{stride}(n) < 0 \\ dy(n) &= j_n - 1 \text{ if } y\text{stride}(n) > 0, \text{ or } dy(n) = j_n - y\text{shape}(n) \text{ if } y\text{stride}(n) < 0 \\ dz(n) &= k_n - 1 \text{ if } z\text{stride}(n) > 0, \text{ or } dz(n) = k_n - z\text{shape}(n) \text{ if } z\text{stride}(n) < 0 \end{aligned}$$

The definitions of indices e , f , and g assume that indexes for arrays x , y , and z are started from unity:

$x(e)$ is defined for $e=1, \dots, \text{length}(x)$
 $y(f)$ is defined for $f=1, \dots, \text{length}(y)$
 $z(g)$ is defined for $g=1, \dots, \text{length}(z)$

Below is a detailed explanation about how elements of the multi-dimensional output vector are stored in the array z for one-dimensional and two-dimensional cases.

One-dimensional case. If $\text{dims}=1$, then $z\text{shape}$ is the number of the output values to be stored in the array z . The actual length of array z may be greater than $z\text{shape}$ elements.

If $z\text{stride} > 1$, output values are stored with the stride: $\text{output}(1)$ is stored to $z(1)$, $\text{output}(2)$ is stored to $z(1+z\text{stride})$, and so on. Hence, the actual length of z must be at least $1+z\text{stride} \cdot (z\text{shape}-1)$ elements or more.

If $z\text{stride} < 0$, it still defines the stride between elements of array z . However, the order of the used elements is the opposite. For the k -th output value, $\text{output}(k)$ is stored in $z(1+|z\text{stride}| \cdot (z\text{shape}-k))$, where $|z\text{stride}|$ is the absolute value of $z\text{stride}$. The actual length of the array z must be at least $1+|z\text{stride}| \cdot (z\text{shape} - 1)$ elements.

Two-dimensional case. If $\text{dims}=2$, the output data is a two-dimensional matrix. The value $z\text{stride}(1)$ defines the stride inside matrix columns, that is, the stride between the $\text{output}(k_1, k_2)$ and $\text{output}(k_1+1, k_2)$ for every pair of indices k_1, k_2 . On the other hand, $z\text{stride}(2)$ defines the stride between columns, that is, the stride between $\text{output}(k_1, k_2)$ and $\text{output}(k_1, k_2+1)$.

If $z\text{stride}(2)$ is greater than $z\text{shape}(1)$, this causes sparse allocation of columns. If the value of $z\text{stride}(2)$ is smaller than $z\text{shape}(1)$, this may result in the transposition of the output matrix. For example, if $z\text{shape} = (2, 3)$, you can define $z\text{stride} = (3, 1)$ to allocate output values like transposed matrix of the shape 3×2 .

Whether $z\text{stride}$ assumes this kind of transformations or not, you need to ensure that different elements $\text{output}(k_1, \dots, k_{\text{dims}})$ will be stored in different locations $z(g)$.

Summary Statistics

The Summary Statistics domain comprises a set of routines that compute basic statistical estimates for single and double precision multi-dimensional datasets.

See the definition of the supported operations in the [Mathematical Notation and Definitions](#) section.

The Summary Statistics routines calculate:

- raw and central moments up to the fourth order
- skewness and excess kurtosis (further referred to as *kurtosis* for brevity)
- variation coefficient
- quantiles and order statistics
- minimum and maximum
- variance-covariance/correlation matrix
- pooled/group variance-covariance matrix and mean
- partial variance-covariance/correlation matrix
- robust estimators for variance-covariance matrix and mean in presence of outliers
- raw/central partial sums up to the fourth order (for brevity referred to as *raw/central sums*)
- matrix of cross-products and sums of squares (for brevity referred to as *cross-product matrix*)
- median absolute deviation, mean absolute deviation

The library also contains functions to perform the following tasks:

- Detect outliers in datasets
- Support missing values in datasets
- Parameterize correlation matrices
- Compute quantiles for streaming data

You can access the Summary Statistics routines through the Fortran 90 and C89 language interfaces. You can use the C89 interface with later versions of the C/C++. You can use the Fortran 90 interface with programs written in Fortran 95.

Intel MKL provides the `mkl_vsl.h` header file for C, and the `mkl_vsl.f90` and `mkl_vsl.f77` header files for Fortran. All header files are in the directory

```
 ${MKL}/include
```

See more details about the Fortran header in the [Random Number Generators](#) section of this chapter.

You can find examples that demonstrate calculation of the Summary Statistics estimates in the examples directories:

```
 ${MKL}/examples/vs1c
```

```
 ${MKL}/examples/vs1f
```

The Summary Statistics API is implemented through task objects, or tasks. A task object is a data structure, or a descriptor, holding parameters that determine a specific Summary Statistics operation. For example, such parameters may be precision, dimensions of user data, the matrix of the observations, or shapes of data arrays.

All the Summary Statistics routines process a task object as follows:

1. Create a task.
2. Modify settings of the task parameters.
3. Compute statistical estimates.
4. Destroy the task.

The Summary Statistics functions fall into the following categories:

Task Constructors - routines that create a new task object descriptor and set up most common parameters (dimension, number of observations, and matrix of the observations).

Task Editors - routines that can set or modify some parameter settings in the existing task descriptor.

Task Computation Routine - a routine that computes specified statistical estimates.

Task Destructor - a routine that deletes the task object and frees the memory.

A Summary Statistics task object contains a series of pointers to the input and output data arrays. You can read and modify the datasets and estimates at any time but you should allocate and release memory for such data.

See detailed information on the algorithms, API, and their usage in the *Intel® MKL Summary Statistics Library Application Notes* [[SSL Notes](#)].

Naming Conventions

The names of the Fortran routines in the Summary Statistics are in lowercase (`vslssreditquantiles`), while the names of types and constants are in uppercase. The names are not case-sensitive.

The names of the C Summary Statistics routines, types, and constants are case-sensitive and can contain lowercase and uppercase characters (`vslsSEditQuantiles`).

The names of routines have the following structure:

`vsl[datatype] SS<base name>` for the C interface

`vsl[datatype] ss<base name>` for the Fortran interface

where

- `vsl` is a prefix indicating that the routine belongs to the Vector Statistical Library of Intel MKL.
- `[datatype]` specifies the type of the input and/or output data and can be `s` (single precision real type), `d` (double precision real type), or `i` (integer type).
- `SS/ss` indicates that the routine is intended for calculations of the Summary Statistics estimates.
- `<base name>` specifies a particular functionality that the routine is designed for, for example, `NewTask`, `Compute`, `DeleteTask`.

NOTE

The Summary Statistics routine `vslDeleteTask` for deletion of the task is independent of the data type and its name omits the `[datatype]` field.

Data Types

The Summary Statistics routines use the following data types for the calculations:

Type	Data Object
<code>Fortran 90: TYPE(VSL_SS_TASK)</code>	Pointer to a Summary Statistics task
<code>C: VSLSSTaskPtr</code>	
<code>Fortran 90: REAL(KIND=4)</code>	Input/output user data in single precision
<code>C: float</code>	
<code>Fortran 90: REAL(KIND=8)</code>	Input/output user data in double precision
<code>C: double</code>	
<code>Fortran 90: INTEGER or INTEGER(KIND=8)</code>	Other data
<code>C: MKL_INT or long long</code>	

NOTE

The actual size of the generic integer type is platform-specific and can be 32 or 64 bits in length. Before you compile your application, set an appropriate size for integers. See details in the 'Using the ILP64 Interface vs. LP64 Interface' section of the *Intel® MKL User's Guide*.

Parameters

The basic parameters in the task descriptor (addresses of dimensions, number of observations, and datasets) are assigned values when the task editors create or modify the task object. Other parameters are determined by the specific task and changed by the task editors.

Task Status and Error Reporting

The task status is an integer value, which is zero if no error is detected, or a specific non-zero error code otherwise. Negative status values indicate errors, and positive values indicate warnings. An error can be caused by invalid parameter values or a memory allocation failure.

The header files define symbolic names for the status codes. These names for the C/C++ interface are defined as macros via the `#define` statements. These names for the Fortran interface are defined as integer constants via the `PARAMETER` operators.

If no error is detected, the function returns the `VSL_STATUS_OK` code, which is defined as zero:

```
C/C++: #define VSL_STATUS_OK 0
F90/F95: INTEGER, PARAMETER::VSL_STATUS_OK = 0
```

In the case of an error, the function returns a non-zero error code that specifies the origin of the failure. The header files define the following status codes for the Summary Statistics error codes:

Summary Statistics Status Codes

Status Code	Description
<code>VSL_STATUS_OK</code>	Operation is successfully completed.
<code>VSL_SS_ERROR_ALLOCATION_FAILURE</code>	Memory allocation has failed.
<code>VSL_SS_ERROR_BAD_DIMEN</code>	Dimension value is invalid.
<code>VSL_SS_ERROR_BAD_OBSERV_N</code>	Invalid number (zero or negative) of observations was obtained.
<code>VSL_SS_ERROR_STORAGE_NOT_SUPPORTED</code>	Storage format is not supported.
<code>VSL_SS_ERROR_BAD_INDC_ADDR</code>	Array of indices is not defined.
<code>VSL_SS_ERROR_BAD_WEIGHTS</code>	Array of weights contains negative values.
<code>VSL_SS_ERROR_BAD_MEAN_ADDR</code>	Array of means is not defined.
<code>VSL_SS_ERROR_BAD_2R_MOM_ADDR</code>	Array of the second order raw moments is not defined.
<code>VSL_SS_ERROR_BAD_3R_MOM_ADDR</code>	Array of the third order raw moments is not defined.
<code>VSL_SS_ERROR_BAD_4R_MOM_ADDR</code>	Array of the fourth order raw moments is not defined.
<code>VSL_SS_ERROR_BAD_2C_MOM_ADDR</code>	Array of the second order central moments is not defined.
<code>VSL_SS_ERROR_BAD_3C_MOM_ADDR</code>	Array of the third order central moments is not defined.
<code>VSL_SS_ERROR_BAD_4C_MOM_ADDR</code>	Array of the fourth order central moments is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_KURTOSIS_ADDR	Array of kurtosis values is not defined.
VSL_SS_ERROR_BAD_SKEWNESS_ADDR	Array of skewness values is not defined.
VSL_SS_ERROR_BAD_MIN_ADDR	Array of minimum values is not defined.
VSL_SS_ERROR_BAD_MAX_ADDR	Array of maximum values is not defined.
VSL_SS_ERROR_BAD_VARIATION_ADDR	Array of variation coefficients is not defined.
VSL_SS_ERROR_BAD_COV_ADDR	Covariance matrix is not defined.
VSL_SS_ERROR_BAD_COR_ADDR	Correlation matrix is not defined.
VSL_SS_ERROR_BAD_QUANT_ORDER_ADDR	Array of quantile orders is not defined.
VSL_SS_ERROR_BAD_QUANT_ORDER	Quantile order value is invalid.
VSL_SS_ERROR_BAD_QUANT_ADDR	Array of quantiles is not defined.
VSL_SS_ERROR_BAD_ORDER_STATS_ADDR	Array of order statistics is not defined.
VSL_SS_ERROR_MOMORDER_NOT_SUPPORTED	Moment of requested order is not supported.
VSL_SS_NOT_FULL_RANK_MATRIX	Correlation matrix is not of full rank.
VSL_SS_ERROR_ALL_OBSERVS_OUTLIERS	All observations are outliers. (At least one observation must not be an outlier.)
VSL_SS_ERROR_BAD_ROBUST_COV_ADDR	Robust covariance matrix is not defined.
VSL_SS_ERROR_BAD_ROBUST_MEAN_ADDR	Array of robust means is not defined.
VSL_SS_ERROR_METHOD_NOT_SUPPORTED	Requested method is not supported.
VSL_SS_ERROR_NULL_TASK_DESCRIPTOR	Task descriptor is null.
VSL_SS_ERROR_BAD_OBSERV_ADDR	Dataset matrix is not defined.
VSL_SS_ERROR_BAD_ACCUM_WEIGHT_ADDR	Pointer to the variable that holds the value of accumulated weight is not defined.
VSL_SS_ERROR_SINGULAR_COV	Covariance matrix is singular.
VSL_SS_ERROR_BAD_POOLED_COV_ADDR	Pooled covariance matrix is not defined.
VSL_SS_ERROR_BAD_POOLED_MEAN_ADDR	Array of pooled means is not defined.
VSL_SS_ERROR_BAD_GROUP_COV_ADDR	Group covariance matrix is not defined.
VSL_SS_ERROR_BAD_GROUP_MEAN_ADDR	Array of group means is not defined.
VSL_SS_ERROR_BAD_GROUP_INDC_ADDR	Array of group indices is not defined.
VSL_SS_ERROR_BAD_GROUP_INDC	Group indices have improper values.
VSL_SS_ERROR_BAD_OUTLIERS_PARAMS_ADDR	Array of parameters for the outlier detection algorithm is not defined.
VSL_SS_ERROR_BAD_OUTLIERS_PARAMS_N_ADDR	Pointer to size of the parameter array for the outlier detection algorithm is not defined.
VSL_SS_ERROR_BAD_OUTLIERS_WEIGHTS_ADDR	Output of the outlier detection algorithm is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_ROBUST_COV_PARAMS_ADDR	Array of parameters of the robust covariance estimation algorithm is not defined.
VSL_SS_ERROR_BAD_ROBUST_COV_PARAMS_N_ADDR	Pointer to the number of parameters of the algorithm for robust covariance is not defined.
VSL_SS_ERROR_BAD_STORAGE_ADDR	Pointer to the variable that holds the storage format is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COV_IDX_ADDR	Array that encodes sub-components of a random vector for the partial covariance algorithm is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COV_IDX	Array that encodes sub-components of a random vector for partial covariance has improper values.
VSL_SS_ERROR_BAD_PARTIAL_COV_ADDR	Partial covariance matrix is not defined.
VSL_SS_ERROR_BAD_PARTIAL_COR_ADDR	Partial correlation matrix is not defined.
VSL_SS_ERROR_BAD_MI_PARAMS_ADDR	Array of parameters for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_PARAMS_N_ADDR	Pointer to number of parameters for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_BAD_PARAMS_N	Size of the parameter array of the Multiple Imputation method is invalid.
VSL_SS_ERROR_BAD_MI_PARAMS	Parameters of the Multiple Imputation method are invalid.
VSL_SS_ERROR_BAD_MI_INIT_ESTIMATES_N_ADDR	Pointer to the number of initial estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_INIT_ESTIMATES_ADDR	Array of initial estimates for the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_ADDR	Array of simulated missing values in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_N_ADDR	Pointer to the size of the array of simulated missing values in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_ESTIMATES_N_ADDR	Pointer to the number of parameter estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_ESTIMATES_ADDR	Array of parameter estimates in the Multiple Imputation method is not defined.
VSL_SS_ERROR_BAD_MI_SIMUL_VALS_N	Invalid size of the array of simulated values in the Multiple Imputation method.
VSL_SS_ERROR_BAD_MI_ESTIMATES_N	Invalid size of an array to hold parameter estimates obtained using the Multiple Imputation method.
VSL_SS_ERROR_BAD_MI_OUTPUT_PARAMS	Array of output parameters in the Multiple Imputation method is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_MI_PRIOR_N_ADDR	Pointer to the number of prior parameters is not defined.
VSL_SS_ERROR_BAD_MI_PRIOR_ADDR	Array of prior parameters is not defined.
VSL_SS_ERROR_BAD_MI_MISSING_VALS_N	Invalid number of missing values was obtained.
VSL_SS_SEMIDEFINITE_COR	Correlation matrix passed into the parameterization function is semi-definite.
VSL_SS_ERROR_BAD_PARAMTR_COR_ADDR	Correlation matrix to be parameterized is not defined.
VSL_SS_ERROR_BAD_COR	All eigenvalues of the correlation matrix to be parameterized are non-positive.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_N_ADDR	Pointer to the number of parameters for the quantile computation algorithm for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_ADDR	Array of parameters of the quantile computation algorithm for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_N	Invalid number of parameters of the quantile computation algorithm for streaming data has been obtained.
VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS	Invalid parameters of the quantile computation algorithm for streaming data have been passed.
VSL_SS_ERROR_BAD_STREAM_QUANT_ORDER_ADDR	Array of the quantile orders for streaming data is not defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_ORDER	Invalid quantile order for streaming data is defined.
VSL_SS_ERROR_BAD_STREAM_QUANT_ADDR	Array of quantiles for streaming data is not defined.
VSL_SS_ERROR_BAD_SUM_ADDR	Array of sums is not defined.
VSL_SS_ERROR_BAD_2R_SUM_ADDR	Array of raw sums of 2nd order is not defined.
VSL_SS_ERROR_BAD_3R_SUM_ADDR	Array of raw sums of 3rd order is not defined.
VSL_SS_ERROR_BAD_4R_SUM_ADDR	Array of raw sums of 4th order is not defined.
VSL_SS_ERROR_BAD_2C_SUM_ADDR	Array of central sums of 2nd order is not defined.
VSL_SS_ERROR_BAD_3C_SUM_ADDR	Array of central sums of 3rd order is not defined.
VSL_SS_ERROR_BAD_4C_SUM_ADDR	Array of central sums of 4th order is not defined.
VSL_SS_ERROR_BAD_CP_SUM_ADDR	Cross-product matrix is not defined.
VSL_SS_ERROR_BAD_MDAD_ADDR	Array of median absolute deviations is not defined.

Status Code	Description
VSL_SS_ERROR_BAD_MNAD_ADDR	Array of mean absolute deviations is not defined.

Routines for robust covariance estimation, outlier detection, partial covariance estimation, multiple imputation, and parameterization of a correlation matrix can return internal error codes that are related to a specific implementation. Such error codes indicate invalid input data or other bugs in the Intel MKL routines other than the Summary Statistics routines.

Task Constructors

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters.

NOTE

If the constructor fails to create a task descriptor, it returns the `NULL` task pointer.

vslSSNewTask

Creates and initializes a new summary statistics task descriptor.

Syntax

Fortran:

```
status = vslsssnewtask(task, p, n, xstorage, x, w, indices)
status = vslldssnewtask(task, p, n, xstorage, x, w, indices)
```

C:

```
status = vslsSSNewTask(&task, p, n, xstorage, x, w, indices);
status = vsldSSNewTask(&task, p, n, xstorage, x, w, indices);
```

Include Files

- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>p</code>	Fortran: INTEGER C: const MKL_INT	Dimension of the task, number of variables
<code>n</code>	Fortran: INTEGER C: const MKL_INT	Number of observations
<code>xstorage</code>	Fortran: INTEGER C: const MKL_INT	Storage format of matrix of observations
<code>x</code>	Fortran: REAL(KIND=4) DIMENSION(*) for <code>vslsssnewtask</code>	Matrix of observations

Name	Type	Description
w	<pre>REAL(KIND=8) DIMENSION(*) for vslDSSNewTask C: const float* for vslsSSNewTask const double* for vslDSSNewTask</pre>	Array of weights of size n . Elements of the arrays are non-negative numbers. If a NULL pointer is passed, each observation is assigned weight equal to 1.
indices	<pre>Fortran: REAL(KIND=4) DIMENSION(*) for vslSSNewTask REAL(KIND=8) DIMENSION(*) for vslDSSNewTask C: const float* for vslsSSNewTask const double* for vslDSSNewTask</pre>	Array of vector components that will be processed. Size of array is p . If a NULL pointer is passed, all components of random vector are processed.

Output Parameters

Name	Type	Description
task	Fortran: TYPE(VSL_SS_TASK) C: VSLSSTaskPtr*	Descriptor of the task
status	Fortran: INTEGER C: int	Set to VSL_STATUS_OK if the task is created successfully, otherwise a non-zero error code is returned.

Description

Each `vslSSNewTask` constructor routine creates a new summary statistics task descriptor with the user-specified value for a required parameter, dimension of the task. The optional parameters (matrix of observations, its storage format, number of observations, weights of observations, and indices of the random vector components) are set to their default values.

The observations of random p -dimensional vector $\xi = (\xi_1, \dots, \xi_i, \dots, \xi_p)$, which are n vectors of dimension p , are passed as a one-dimensional array x . The parameter `xstorage` defines the storage format of the observations and takes one of the possible values listed in [Table "Storage format of matrix of observations and order statistics"](#).

Storage format of matrix of observations and order statistics

Parameter	Description
<code>VSL_SS_MATRIX_STORAGE_ROWS</code>	The observations of random vector ξ are packed by rows: n data points for the vector component ξ_1 come first, n data points for the vector component ξ_2 come second, and so forth.
<code>VSL_SS_MATRIX_STORAGE_COLS</code>	The observations of random vector ξ are packed by columns: the first p -dimensional observation of the vector ξ comes first, the second p -dimensional observation of the vector comes second, and so forth.

NOTE

Since matrices in Fortran are stored by columns while in C they are stored by rows, initialization of the `xstorage` variable in Fortran is opposite to that in C. Set `xstorage` to `VSL_SS_MATRIX_STORAGE_COLS`, if the dataset is stored as a two-dimensional matrix that consists of p rows and n columns; otherwise, use the `VSL_SS_MATRIX_STORAGE_ROWS` constant.

A one-dimensional array `w` of size n contains non-negative weights assigned to the observations. You can pass a `NULL` array into the constructor. In this case, each observation is assigned the default value of the weight.

You can choose vector components for which you wish to compute statistical estimates. If an element of the vector indices of size p contains 0, the observations that correspond to this component are excluded from the calculations. If you pass the `NULL` value of the parameter into the constructor, statistical estimates for all random variables are computed.

If the constructor fails to create a task descriptor, it returns the `NULL` task pointer.

Task Editors

Task editors are intended to set up or change the task parameters listed in [Table "Parameters of Summary Statistics Task to Be Initialized or Modified"](#). As an example, to compute the sample mean for a one-dimensional dataset, initialize a variable for the mean value, and pass its address into the task as shown in the example below:

```
#define DIM    1
#define N     1000

int main()
{
    VSLSSTaskPtr task;
    double x[N];
    double mean;
    MKL_INT p, n, xstorage;
    int status;
    /* initialize variables used in the computations of sample mean */
    p = DIM;
    n = N;
    xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
    mean = 0.0;

    /* create task */
    status = vslsSsNewTask( &task, &p, &n, &xstorage, x, 0, 0 );

    /* initialize task parameters */
    status = vslsSsEditTask( task, VSL_SS_ED_MEAN, &mean );

    /* compute mean using SS fast method */
    status = vslsSsCompute(task, VSL_SS_MEAN, VSL_SS_METHOD_FAST );

    /* deallocate task resources */
    status = vslsSsDeleteTask( &task );

    return 0;
}
```

Use the single (`vslsssedittask`) or double (`vslsSseditTask`) version of an editor, to initialize single or double precision version task parameters, respectively. Use an integer version of an editor (`vslsSseditTask`) to initialize parameters of the integer type.

Table "Summary Statistics Task Editors" lists the task editors for Summary Statistics. Each of them initializes and/or modifies a respective group of related parameters.

Summary Statistics Task Editors

Editor	Description
<code>vslSSEditTask</code>	Changes a pointer in the task descriptor.
<code>vslSSEditMoments</code>	Changes pointers to arrays associated with raw and central moments.
<code>vslSSEditSums</code>	Modifies the pointers to arrays that hold sum estimates.
<code>vslSSEditCovCor</code>	Changes pointers to arrays associated with covariance and/or correlation matrices.
<code>vslSSEditCP</code>	Modifies the pointers to cross-product matrix parameters.
<code>vslSSEditPartialCovCor</code>	Changes pointers to arrays associated with partial covariance and/or correlation matrices.
<code>vslSSEditQuantiles</code>	Changes pointers to arrays associated with quantile/order statistics calculations.
<code>vslSSEditStreamQuantiles</code>	Changes pointers to arrays for quantile related calculations for streaming data.
<code>vslSSEditPooledCovariance</code>	Changes pointers to arrays associated with algorithms related to a pooled covariance matrix.
<code>vslSSEditRobustCovariance</code>	Changes pointers to arrays for robust estimation of a covariance matrix and mean.
<code>vslSSEditOutliersDetection</code>	Changes pointers to arrays for detection of outliers.
<code>vslSSEditMissingValues</code>	Changes pointers to arrays associated with the method of supporting missing values in a dataset.
<code>vslSSEditCorParameterization</code>	Changes pointers to arrays associated with the algorithm for parameterization of a correlation matrix.

NOTE

You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

`vslSSEditTask`

Modifies address of an input/output parameter in the task descriptor.

Syntax

Fortran:

```
status = vslsssedittask(task, parameter, par_addr)
status = vslssedittask(task, parameter, par_addr)
status = vslissedittask(task, parameter, par_addr)
```

C:

```
status = vslsSSEditTask(task, parameter, par_addr);
status = vsldSSEditTask(task, parameter, par_addr);
```

```
status = vslisSEditTask(task, parameter, par_addr);
```

Include Files

- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
task	Fortran: TYPE(VSL_SS_TASK) C: VSLSSTaskPtr	Descriptor of the task
parameter	Fortran: INTEGER C: const MKL_INT	Parameter to change
par_addr	Fortran: REAL(KIND=4) DIMENSION(*) for vslsssedittask REAL(KIND=8) DIMENSION(*) for vsldssedittask INTEGER DIMENSION(*) for vslissedittask C: const float* for vslsSEditTask const double* for vsldSSEditTask const MKL_INT* for vslisSEditTask	Address of the new parameter

Output Parameters

Name	Type	Description
status	Fortran: INTEGER C: int	Current status of the task

Description

The `vslsSEditTask` routine replaces the pointer to the parameter stored in the Summary Statistics task descriptor with the `par_addr` pointer. If you pass the NULL pointer to the editor, no changes take place in the task and a corresponding error code is returned. See [Table "Parameters of Summary Statistics Task to Be Initialized or Modified"](#) for the predefined values of the parameter.

Use the single (`vslssedittask`) or double (`vsldssedittask`) version of the editor, to initialize single or double precision version task parameters, respectively. Use an integer version of the editor (`vslissedittask`) to initialize parameters of the integer type.

Parameters of Summary Statistics Task to Be Initialized or Modified

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_DIMEN	i	Address of a variable that holds the task dimension	Required. Positive integer value.
VSL_SS_ED_OBSERV_N	i	Address of a variable that holds the number of observations	Required. Positive integer value.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_OBSERV	d, s	Address of the observation matrix	Required. Provide the matrix containing your observations.
VSL_SS_ED_OBSERV_STORAGE	i	Address of a variable that holds the storage format for the observation matrix	Required. Provide a storage format supported by the library whenever you pass a matrix of observations. ¹
VSL_SS_ED_INDC	i	Address of the array of indices	Optional. Provide this array if you need to process individual components of the random vector. Set entry i of the array to one to include the i th coordinate in the analysis. Set entry i of the array to zero to exclude the i th coordinate from the analysis.
VSL_SS_ED_WEIGHTS	d, s	Address of the array of observation weights	Optional. If the observations have weights different from the default weight (one), set entries of the array to non-negative floating point values.
VSL_SS_ED_MEAN	d, s	Address of the array of means	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2R_MOM	d, s	Address of an array of raw moments of the second order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3R_MOM	d, s	Address of an array of raw moments of the third order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_4R_MOM	d, s	Address of an array of raw moments of the fourth order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2C_MOM	d, s	Address of an array of central moments of the second order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first and second order.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_3C_MOM	d, s	Address of an array of central moments of the third order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, and third order.
VSL_SS_ED_4C_MOM	d, s	Address of an array of central moments of the fourth order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, third, and fourth order.
VSL_SS_ED_KURTOSIS	d, s	Address of the array of kurtosis estimates	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, third, and fourth order.
VSL_SS_ED_SKEWNESS	d, s	Address of the array of skewness estimates	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, and third order.
VSL_SS_ED_MIN	d, s	Address of the array of minimum estimates	Optional. Set entries of array to meaningful values, such as the values of the first observation.
VSL_SS_ED_MAX	d, s	Address of the array of maximum estimates	Optional. Set entries of array to meaningful values, such as the values of the first observation.
VSL_SS_ED_VARIATION	d, s	Address of the array of variation coefficients	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first and second order.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_COV	d, s	Address of a covariance matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Make sure you also provide an array for the mean.
VSL_SS_ED_COV_STORAGE	i	Address of the variable that holds the storage format for a covariance matrix	Required. Provide a storage format supported by the library whenever you intend to compute the covariance matrix. ²
VSL_SS_ED_COR	d, s	Address of a correlation matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. If you initialize the matrix in non-trivial way, make sure that the main diagonal contains variance values. Also, provide an array for the mean.
VSL_SS_ED_COR_STORAGE	i	Address of the variable that holds the correlation storage format for a correlation matrix	Required. Provide a storage format supported by the library whenever you intend to compute the correlation matrix. ²
VSL_SS_ED_ACCUM_WEIGHT	d, s	Address of the array of size 2 that holds the accumulated weight (sum of weights) in the first position and the sum of weights squared in the second position	Optional. Set the entries of the matrix to meaningful values (typically zero) if you intend to do progressive processing of the dataset or need the sum of weights and sum of squared weights assigned to observations.
VSL_SS_ED_QUANT_ORDER_N	i	Address of the variable that holds the number of quantile orders	Required. Positive integer value. Provide the number of quantile orders whenever you compute quantiles.
VSL_SS_ED_QUANT_ORDER	d, s	Address of the array of quantile orders	Required. Set entries of array to values from the interval (0,1). Provide this parameter whenever you compute quantiles.
VSL_SS_ED_QUANT_QUANTILES	d, s	Address of the array of quantiles	None.
VSL_SS_ED_ORDER_STATS	d, s	Address of the array of order statistics	None.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_GROUP_INDC	i	Address of the array of group indices used in computation of a pooled covariance matrix	Required. Set entry i to integer value k if the observation belongs to group k . Values of k take values in the range $[0, g-1]$, where g is the number of groups.
VSL_SS_ED_POOLED_COV_STORAGE		Address of a variable that holds the storage format for a pooled covariance matrix	Required. Provide a storage format supported by the library whenever you intend to compute pooled covariance. ²
VSL_SS_ED_POOLED_MEAN	d, s	Address of an array of pooled means	None.
VSL_SS_ED_POOLED_COV	d, s	Address of pooled covariance matrices	None.
VSL_SS_ED_GROUP_COV_INDC	i	Address of an array of indices for which covariance/means should be computed	Optional. Set the k th entry of the array to 1 if you need group covariance and mean for group k ; otherwise set it to zero.
VSL_SS_ED_REQ_GROUP_INDC	i	Address of an array of indices for which group estimates such as covariance or means are requested	Optional. Set the k th entry of the array to 1 if you need an estimate for group k ; otherwise set it to zero.
VSL_SS_ED_GROUP_MEANS	i	Address of an array of group means	None.
VSL_SS_ED_GROUP_COV_STORAGE	d, s	Address of a variable that holds the storage format for a group covariance matrix	Required. Provide a storage format supported by the library whenever you intend to get group covariance. ²
VSL_SS_ED_GROUP_COV	d, s	Address of group covariance matrices	None.
VSL_SS_ED_ROBUST_COV_STORAGE	s	Address of a variable that holds the storage format for a robust covariance matrix	Required. Provide a storage format supported by the library whenever you compute robust covariance ² .
VSL_SS_ED_ROBUST_COV_PARAMS_N		Address of a variable that holds the number of algorithmic parameters of the method for robust covariance estimation	Required. Set to the number of TBS parameters, <i>VSL_SS_TBS_PARAMS_N</i> .
VSL_SS_ED_ROBUST_COV_PARAMS	s	Address of an array of parameters of the method for robust estimation of a covariance	Required. Set the entries of the array according to the description in vsISSEditRobustCovariance .

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_ROBUST_MEAN	i	Address of an array of robust means	None.
VSL_SS_ED_ROBUST_COV	d, s	Address of a robust covariance matrix	None.
VSL_SS_ED_OUTLIERS_PARAMS_N	i, s	Address of a variable that holds the number of parameters of the outlier detection method	Required. Set to the number of outlier detection parameters, <i>VSL_SS_BACON_PARAMS_N</i> .
VSL_SS_ED_OUTLIERS_PARAMS	i	Address of an array of algorithmic parameters for the outlier detection method	Required. Set the entries of the array according to the description in vslSSEditOutliersDetection .
VSL_SS_ED_OUTLIERS_WEIGHT	d, s	Address of an array of weights assigned to observations by the outlier detection method	None.
VSL_SS_ED_ORDER_STATS_STORAGE	i	Address of a variable that holds the storage format of an order statistics matrix	Required. Provide a storage format supported by the library whenever you compute a matrix of order statistics. ¹
VSL_SS_ED_PARTIAL_COV_IDX	i	Address of an array that encodes subcomponents of a random vector	Required. Set the entries of the array according to the description in vslSSEditPartialCovCor .
VSL_SS_ED_PARTIAL_COV	d, s	Address of a partial covariance matrix	None.
VSL_SS_ED_PARTIAL_COV_STORAGE	i	Address of a variable that holds the storage format of a partial covariance matrix	Required. Provide a storage format supported by the library whenever you compute the partial covariance. ²
VSL_SS_ED_PARTIAL_COR	d, s	Address of a partial correlation matrix	None.
VSL_SS_ED_PARTIAL_COR_STORAGE	i	Address of a variable that holds the storage format for a partial correlation matrix	Required. Provide a storage format supported by the library whenever you compute the partial correlation. ²
VSL_SS_ED_MI_PARAMS_N	i	Address of a variable that holds the number of algorithmic parameters for the Multiple Imputation method	Required. Set to the number of MI parameters, <i>VSL_SS_MI_PARAMS_SIZE</i> .
VSL_SS_ED_MI_PARAMS	d, s	Address of an array of algorithmic parameters for the Multiple Imputation method	Required. Set entries of the array according to the description in vslSSEditMissingValues .

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_MI_INIT_ESTIMATES_N		Address of a variable that holds the number of initial estimates for the Multiple Imputation method	Optional. Set to $p+p*(p+1)/2$, where p is the task dimension.
VSL_SS_ED_MI_INIT_ESTIMATES, s		Address of an array of initial estimates for the Multiple Imputation method	Optional. Set the values of the array according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics Library" in the <i>Intel® MKL Summary Statistics Library Application Notes</i> document [SSL Notes].
VSL_SS_ED_MI_SIMUL_VALS_N i		Address of a variable that holds the number of simulated values in the Multiple Imputation method	Optional. Positive integer indicating the number of missing points in the observation matrix.
VSL_SS_ED_MI_SIMUL_VALS d, s		Address of an array of simulated values in the Multiple Imputation method	None.
VSL_SS_ED_MI_ESTIMATES_N i		Address of a variable that holds the number of estimates obtained as a result of the Multiple Imputation method	Optional. Positive integer number defined according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics Library" in the <i>Intel® MKL Summary Statistics Library Application Notes</i> document [SSL Notes].
VSL_SS_ED_MI_ESTIMATES d, s		Address of an array of estimates obtained as a result of the Multiple Imputation method	None.
VSL_SS_ED_MI_PRIOR_N i		Address of a variable that holds the number of prior parameters for the Multiple Imputation method	Optional. If you pass a user-defined array of prior parameters, set this parameter to $(p^2+3*p+4)/2$, where p is the task dimension.
VSL_SS_ED_MI_PRIOR d, s		Address of an array of prior parameters for the Multiple Imputation method	Optional. Set entries of the array of prior parameters according to the description in "Basic Components of the Multiple Imputation Function in Summary Statistics Library" in the <i>Intel® MKL Summary Statistics Library Application Notes</i> document [SSL Notes].

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_PARAMTR_COR	d, s	Address of a parameterized correlation matrix	None.
VSL_SS_ED_PARAMTR_COR_STORAGE		Address of a variable that holds the storage format of a parameterized correlation matrix	Required. Provide a storage format supported by the library whenever you compute the parameterized correlation matrix. ²
VSL_SS_ED_STREAM_QUANT_PARAMS_N		Address of a variable that holds the number of parameters of a quantile computation method for streaming data	Required. Set to the number of quantile computation parameters, <i>VSL_SS_SQUANTS_ZW_PARAMS_N</i> .
VSL_SS_ED_STREAM_QUANT_PARAMS		Address of an array of parameters of a quantile computation method for streaming data	Required. Set the entries of the array according to the description in "Computing Quantiles for Streaming Data" in the <i>Intel® MKL Summary Statistics Library Application Notes</i> document [SSL Notes].
VSL_SS_ED_STREAM_QUANT_ORDER_N		Address of a variable that holds the number of quantile orders for streaming data	Required. Positive integer value.
VSL_SS_ED_STREAM_QUANT_ORDER_S	s	Address of an array of quantile orders for streaming data	Required. Set entries of the array to values from the interval (0,1). Provide this parameter whenever you compute quantiles.
VSL_SS_ED_STREAM_QUANT_QUANTILES		Address of an array of quantiles for streaming data	None.
VSL_SS_ED_SUM	d, s	Address of array of sums	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2R_SUM	d, s	Address of array of raw sums of 2nd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3R_SUM	d, s	Address of array of raw sums of 3rd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.

Parameter Value	Type	Purpose	Initialization
VSL_SS_ED_4R_SUM	d, s	Address of array of raw sums of 4th order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_2C_SUM	d, s	Address of array of central sums of 2nd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_3C_SUM	d, s	Address of array of central sums of 3rd order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_4C_SUM	d, s	Address of array of central sums of 4th order	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_CP	d, s	Address of cross-product matrix	Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.
VSL_SS_ED_MDAD	d, s	Address of array of median absolute deviations	None.
VSL_SS_ED_MNAD	d, s	Address of array of mean absolute deviations	None.

1. See [Table: "Storage format of matrix of observations and order statistics"](#) for storage formats.
2. See [Table: "Storage formats of a variance-covariance/correlation matrix"](#) for storage formats.

vslSSSEditMoments

Modifies the pointers to arrays that hold moment estimates.

Syntax

Fortran:

```
status = vslssreditmoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m)
status = vslssreditmoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m)
```

C:

```
status = vslsSSEditMoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m);
status = vslsSSEditMoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m);
```

Include Files

- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE (VSL_SS_TASK) C: VSLSSTaskPtr	Descriptor of the task
<i>mean</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vsldsseditmoments REAL (KIND=8) DIMENSION(*) for vsldsseditmoments C: const float* for vsldssEditMoments const double* for vsldSSEditMoments	Pointer to the array of means
<i>r2m</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vsldsseditmoments REAL (KIND=8) DIMENSION(*) for vsldsseditmoments C: const float* for vsldssEditMoments const double* for vsldSSEditMoments	Pointer to the array of raw moments of the 2 nd order
<i>r3m</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vsldsseditmoments REAL (KIND=8) DIMENSION(*) for vsldsseditmoments C: const float* for vsldssEditMoments const double* for vsldSSEditMoments	Pointer to the array of raw moments of the 3 rd order
<i>r4m</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vsldsseditmoments REAL (KIND=8) DIMENSION(*) for vsldsseditmoments C: const float* for vsldssEditMoments const double* for vsldSSEditMoments	Pointer to the array of raw moments of the 4 th order
<i>c2m</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vsldsseditmoments REAL (KIND=8) DIMENSION(*) for vsldsseditmoments	Pointer to the array of central moments of the 2 nd order

Name	Type	Description
<i>c3m</i>	<pre>C: const float* for vslssSEEditMoments const double* for vslsSSEEditMoments</pre> <p>Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditmoments</p> <pre>REAL(KIND=8) DIMENSION(*) for vslsSSEEditMoments C: const float* for vslsSSEEditMoments const double* for vslsSSEEditMoments</pre>	Pointer to the array of central moments of the 3 rd order
<i>c4m</i>	<pre>REAL(KIND=4) DIMENSION(*) for vslsseditmoments</pre> <p>Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditmoments</p> <pre>REAL(KIND=8) DIMENSION(*) for vslsSSEEditMoments C: const float* for vslsSSEEditMoments const double* for vslsSSEEditMoments</pre>	Pointer to the array of central moments of the 4 th order

Output Parameters

Name	Type	Description
<i>status</i>	<p>Fortran: INTEGER</p> <p>C: int</p>	Current status of the task

Description

The `vslssSEEditMoments` routine replaces pointers to the arrays that hold estimates of raw and central moments with values passed as corresponding parameters of the routine. If an input parameter is `NULL`, the value of the relevant parameter remains unchanged.

vslssSEEditSums

Modifies the pointers to arrays that hold sum estimates.

Syntax

Fortran:

```
status = vslssseditsums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s)
status = vslsSSEEditSums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s)
```

C:

```
status = vslssSEEditSums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s);
status = vslsSSEEditSums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s);
```

Include Files

- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE (VSL_SS_TASK) C: VSLSSTaskPtr	Descriptor of the task
<i>sum</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vslsseditsums REAL (KIND=8) DIMENSION(*) for vsldsseditsums C: const float* for vslsSSEditSums const double* for vsldSSEditSums	Pointer to the array of sums
<i>r2s</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vslsseditsums REAL (KIND=8) DIMENSION(*) for vsldsseditsums C: const float* for vslsSSEditSums const double* for vsldSSEditSums	Pointer to the array of raw sums of the second order
<i>r3s</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vslsseditsums REAL (KIND=8) DIMENSION(*) for vsldsseditsums C: const float* for vslsSSEditSums const double* for vsldSSEditSums	Pointer to the array of raw sums of the third order
<i>r4s</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vslsseditsums REAL (KIND=8) DIMENSION(*) for vsldsseditsums C: const float* for vslsSSEditSums const double* for vsldSSEditSums	Pointer to the array of raw sums of the fourth order
<i>c2s</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vslsseditsums REAL (KIND=8) DIMENSION(*) for vsldsseditsums C: const float* for vslsSSEditSums const double* for vsldSSEditSums	Pointer to the array of central sums of the second order

Name	Type	Description
<i>c3s</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vs1ssseditsums C: const float* for vs1sSSEditSums const double* for vs1dSSEditSums	Pointer to the array of central sums of the third order
<i>c4s</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vs1ssseditsums C: const float* for vs1sSSEditSums const double* for vs1dSSEditSums	Pointer to the array of central sums of the fourth order

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Current status of the task

Description

The `vslSSEditSums` routine replaces pointers to the arrays that hold estimates of raw and central sums with values passed as corresponding parameters of the routine. If an input parameter is `NULL`, the value of the relevant parameter remains unchanged.

`vslSSEditCovCor`

Modifies the pointers to covariance/correlation/cross-product parameters.

Syntax

Fortran:

```
status = vslsseditcovcor(task, mean, cov, cov_storage, cor, cor_storage)
status = vsldsseditcovcor(task, mean, cov, cov_storage, cor, cor_storage)
```

C:

```
status = vs1sSSEditCovCor(task, mean, cov, cov_storage, cor, cor_storage);
status = vs1dSSEditCovCor(task, mean, cov, cov_storage, cor, cor_storage);
```

Include Files

- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE (VSL_SS_TASK) C: VSLSSTaskPtr	Descriptor of the task
<i>mean</i>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditcovcor REAL (KIND=8) DIMENSION (*) for vsldsseditcovcor C: const float* for vslsSSEditCovCor const double* for vsldSSEditCovCor	Pointer to the array of means
<i>cov</i>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditcovcor REAL (KIND=8) DIMENSION (*) for vsldsseditcovcor C: const float* for vslsSSEditCovCor const double* for vsldSSEditCovCor	Pointer to a covariance matrix
<i>cov_storage</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the storage format of the covariance matrix
<i>cor</i>	Fortran: REAL (KIND=4) DIMENSION (*) for vslsseditcovcor REAL (KIND=8) DIMENSION (*) for vsldsseditcovcor C: const float* for vslsSSEditCovCor const double* for vsldSSEditCovCor	Pointer to a correlation matrix
<i>cor_storage</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the storage format of the correlation matrix

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Current status of the task

Description

The `vslSSEditCovCor` routine replaces pointers to the array of means, covariance/correlation arrays, and their storage format with values passed as corresponding parameters of the routine. See [Table "Storage formats of a variance-covariance/correlation/cross-product matrix"](#) for possible values of the `cov_storage` and `cor_storage` parameters. If an input parameter is `NULL`, the old value of the parameter remains unchanged in the Summary Statistics task descriptor.

[Storage formats of a variance-covariance/correlation/cross-product matrix](#)

Parameter	Description
<code>VSL_SS_MATRIX_STORAGE_FULL</code>	A symmetric variance-covariance/correlation/cross-product matrix is a one-dimensional array with elements $c(i, j)$ stored as $cp(i*p + j)$. The size of the array is $p*p$.
<code>VSL_SS_MATRIX_STORAGE_L_PACKED</code>	A symmetric variance-covariance/correlation/cross-product matrix with elements $c(i, j)$ is packed as a one-dimensional array $cp(i + (2n - j)*(j - 1)/2)$ for $j \leq i$. The size of the array is $p*(p+1)/2$.
<code>VSL_SS_MATRIX_STORAGE_U_PACKED</code>	A symmetric variance-covariance/correlation/cross-product matrix with elements $c(i, j)$ is packed as a one-dimensional array $cp(i + j*(j - 1)/2)$ for $i \leq j$. The size of the array is $p*(p+1)/2$.

`vslSSEditCP`

Modifies the pointers to cross-product matrix parameters.

Syntax

Fortran:

```
status = vslssseditcp(task, mean, sum, cp, cp_storage)
status = vslrsseditcp(task, mean, sum, cp, cp_storage)
```

C:

```
status = vslsSSEditCP(task, mean, sum, cp, cp_storage);
status = vslsSSEditCP(task, mean, sum, cp, cp_storage);
```

Include Files

- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	Fortran: <code>TYPE(VSL_SS_TASK)</code> C: <code>VSLSSTaskPtr</code>	Descriptor of the task
<code>mean</code>	Fortran: <code>REAL(KIND=4) DIMENSION(*)</code> for <code>vslssseditcp</code>	Pointer to array of means

Name	Type	Description
	REAL(KIND=8) DIMENSION(*) for vslsseditcp C: const float* for vs1sSSEditCP const double* for vsldSSEditCP	
sum	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditcp REAL(KIND=8) DIMENSION(*) for vslsseditcp C: const float* for vs1sSSEditCP const double* for vsldSSEditCP	Pointer to array of sums
cp	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditcp REAL(KIND=8) DIMENSION(*) for vslsseditcp C: const float* for vs1sSSEditCP const double* for vsldSSEditCP	Pointer to a cross-product matrix
cp_storage	Fortran: INTEGER C: const MKL_INT*	Pointer to the storage format of the cross-product matrix

Output Parameters

Name	Type	Description
status	Fortran: INTEGER C: int	Current status of the task

Description

The `vslsSEditCP` routine replaces pointers to the array of means, array of sums, cross-product matrix, and its storage format with values passed as corresponding parameters of the routine. See [Table: "Storage formats of a variance-covariance/correlation/cross-product matrix"](#) for possible values of the `cp_storage` parameter. If an input parameter is `NULL`, the old value of the parameter remains unchanged in the Summary Statistics task descriptor.

Storage formats of a variance-covariance/correlation/cross-product matrix

Parameter	Description
<code>VSL_SS_MATRIX_STORAGE_FULL</code>	A symmetric variance-covariance/correlation/cross-product matrix is a one-dimensional array with elements $c(i, j)$ stored as $cp(i*p + j)$. The size of the array is $p*p$.
<code>VSL_SS_MATRIX_STORAGE_L_PACKED</code>	A symmetric variance-covariance/correlation/cross-product matrix with elements $c(i, j)$ is packed as a one-dimensional array $cp(i + (2n - j)*(j - 1)/2)$ for $j \leq i$. The size of the array is $p*(p+1)/2$.

Parameter	Description
VSL_SS_MATRIX_STORAGE_U_PACKED	A symmetric variance-covariance/correlation/cross-product matrix with elements $c(i, j)$ is packed as a one-dimensional array $cp(i + j*(j - 1)/2)$ for $i \leq j$. The size of the array is $p*(p + 1)/2$.

vslSSEditPartialCovCor

Modifies the pointers to partial covariance/correlation parameters.

Syntax

Fortran:

```
status = vslssreditpartialcovcor(task, p_idx_array, cov, cov_storage, cor,
cor_storage, p_cov, p_cov_storage, p_cor, p_cor_storage)
```

```
status = vslrssreditpartialcovcor(task, p_idx_array, cov, cov_storage, cor,
cor_storage, p_cov, p_cov_storage, p_cor, p_cor_storage)
```

C:

```
status = vslsSSEditPartialCovCor(task, p_idx_array, cov, cov_storage, cor,
cor_storage, p_cov, p_cov_storage, p_cor, p_cor_storage);
```

```
status = vslsSSEditPartialCovCor(task, p_idx_array, cov, cov_storage, cor,
cor_storage, p_cov, p_cov_storage, p_cor, p_cor_storage);
```

Include Files

- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
task	Fortran: TYPE(VSL_SS_TASK) C: VSLSTaskPtr	Descriptor of the task
p_idx_array	Fortran: INTEGER C: const MKL_INT*	Pointer to the array that encodes indices of subcomponents Z and Y of the random vector as described in section Mathematical Notation and Definitions . $p_{idx_array}[i]$ equals to -1 if the i -th component of the random vector belongs to Z 1, if the i -th component of the random vector belongs to Y.
cov	Fortran: REAL(KIND=4) DIMENSION(*) for vslssreditpartialcovcor REAL(KIND=8) DIMENSION(*) for vslrssreditpartialcovcor	Pointer to a covariance matrix

Name	Type	Description
	<pre>C: const float* for vslsSSEditPartialCovCor const double* for vsldSSEditPartialCovCor</pre>	
<i>cov_storage</i>	Fortran: INTEGER <pre>C: const MKL_INT*</pre>	Pointer to the storage format of the covariance matrix
<i>cor</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditpartialcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditpartialcovcor C: const float* for vslsSSEditPartialCovCor const double* for vsldSSEditPartialCovCor	Pointer to a correlation matrix
<i>cor_storage</i>	Fortran: INTEGER <pre>C: const MKL_INT*</pre>	Pointer to the storage format of the correlation matrix
<i>p_cov</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditpartialcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditpartialcovcor C: const float* for vslsSSEditPartialCovCor const double* for vsldSSEditPartialCovCor	Pointer to a partial covariance matrix
<i>p_cov_storage</i>	Fortran: INTEGER <pre>C: const MKL_INT*</pre>	Pointer to the storage format of the partial covariance matrix
<i>p_cor</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditpartialcovcor REAL(KIND=8) DIMENSION(*) for vsldsseditpartialcovcor C: const float* for vslsSSEditPartialCovCor const double* for vsldSSEditPartialCovCor	Pointer to a partial correlation matrix
<i>p_cor_storage</i>	Fortran: INTEGER <pre>C: const MKL_INT*</pre>	Pointer to the storage format of the partial correlation matrix

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Current status of the task

Description

The `vslSSEditPartialCovCor` routine replaces pointers to covariance/correlation arrays, partial covariance/correlation arrays, and their storage format with values passed as corresponding parameters of the routine. See [Table "Storage formats of a variance-covariance/correlation matrix"](#) for possible values of the `cov_storage`, `cor_storage`, `p_cov_storage`, and `p_cor_storage` parameters. If an input parameter is `NULL`, the old value of the parameter remains unchanged in the Summary Statistics task descriptor.

vslSSEditQuantiles

Modifies the pointers to parameters related to quantile computations.

Syntax

Fortran:

```
status = vslsseditquantiles(task, quant_order_n, quant_order, quants, order_stats,
order_stats_storage)

status = vslseditquantiles(task, quant_order_n, quant_order, quants, order_stats,
order_stats_storage)
```

C:

```
status = vslSSEditQuantiles(task, quant_order_n, quant_order, quants, order_stats,
order_stats_storage);

status = vslsSEditQuantiles(task, quant_order_n, quant_order, quants, order_stats,
order_stats_storage);
```

Include Files

- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(VSL_SS_TASK) C: VSLSTaskPtr	Descriptor of the task
<i>quant_order_n</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the number of quantile orders
<i>quant_order</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditquantiles REAL(KIND=8) DIMENSION(*) for vslseditquantiles	Pointer to the array of quantile orders

Name	Type	Description
	C: const float* for vslssSEditQuantiles const double* for vsldSSEditQuantiles	
quants	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditquantiles REAL(KIND=8) DIMENSION(*) for vsldsseditquantiles C: const float* for vslssSEditQuantiles const double* for vsldSSEditQuantiles	Pointer to the array of quantiles
order_stats	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditquantiles REAL(KIND=8) DIMENSION(*) for vsldsseditquantiles C: const float* for vslssSEditQuantiles const double* for vsldSSEditQuantiles	Pointer to the array of order statistics
order_stats_storage	Fortran: INTEGER C: const MKL_INT*	Pointer to the storage format of the order statistics array

Output Parameters

Name	Type	Description
status	Fortran: INTEGER C: int	Current status of the task

Description

The `vslssSEditQuantiles` routine replaces pointers to the number of quantile orders, the array of quantile orders, the array of quantiles, the array that holds order statistics, and the storage format for the order statistics with values passed into the routine. See [Table "Storage format of matrix of observations and order statistics"](#) for possible values of the `order_statistics_storage` parameter. If an input parameter is `NULL`, the corresponding parameter in the Summary Statistics task descriptor remains unchanged.

`vslssSEditStreamQuantiles`

Modifies the pointers to parameters related to quantile computations for streaming data.

Syntax

Fortran:

```
status = vslsseditstreamquantiles(task, quant_order_n, quant_order, quants, nparams,  
params)
```

```
status = vslsSSEditStreamQuantiles(task, quant_order_n, quant_order, quants, nparams,
params)
```

C:

```
status = vsldsseditstreamquantiles(task, quant_order_n, quant_order, quants, nparams,
params);
```

```
status = vsldSSEditStreamQuantiles(task, quant_order_n, quant_order, quants, nparams,
params);
```

Include Files

- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(VSL_SS_TASK) C: VSLSSTaskPtr	Descriptor of the task
<i>quant_order_n</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the number of quantile orders
<i>quant_order</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsSSEditStreamQuantiles REAL(KIND=8) DIMENSION(*) for vsldsseditstreamquantiles C: const float* for vslsSSEditStreamQuantiles const double* for vsldSSEditStreamQuantiles	Pointer to the array of quantile orders
<i>quants</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsSSEditStreamQuantiles REAL(KIND=8) DIMENSION(*) for vsldsseditstreamquantiles C: const float* for vslsSSEditStreamQuantiles const double* for vsldSSEditStreamQuantiles	Pointer to the array of quantiles
<i>nparams</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the number of the algorithm parameters
<i>params</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsSSEditStreamQuantiles REAL(KIND=8) DIMENSION(*)	Pointer to the array of the algorithm parameters

Name	Type	Description
	<code>for vslsSSEditStreamQuantiles</code>	
	<code>C: const float*</code>	
	<code>for vslsSSEditStreamQuantiles</code>	
	<code>const double*</code>	
	<code>for vslsSSEditStreamQuantiles</code>	

Output Parameters

Name	Type	Description
<code>status</code>	Fortran: INTEGER C: int	Current status of the task

Description

The `vslsSSEditStreamQuantiles` routine replaces pointers to the number of quantile orders, the array of quantile orders, the array of quantiles, the number of the algorithm parameters, and the array of the algorithm parameters with values passed into the routine. If an input parameter is `NULL`, the corresponding parameter in the Summary Statistics task descriptor remains unchanged.

vslsSSEditPooledCovariance

Modifies pooled/group covariance matrix array pointers.

Syntax

Fortran:

```
status = vslsSSEditPooledCovariance(task, grp_indices, pld_mean, pld_cov,
req_grp_indices, grp_means, grp_cov)
status = vslsSSEditPooledCovariance(task, grp_indices, pld_mean, pld_cov,
req_grp_indices, grp_means, grp_cov)
```

C:

```
status = vslsSSEditPooledCovariance(task, grp_indices, pld_mean, pld_cov,
req_grp_indices, grp_means, grp_cov);
status = vslsSSEditPooledCovariance(task, grp_indices, pld_mean, pld_cov,
req_grp_indices, grp_means, grp_cov);
```

Include Files

- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	Fortran: TYPE(VSL_SS_TASK) C: VSLSTaskPtr	Descriptor of the task

Name	Type	Description
<i>grp_indices</i>	Fortran: INTEGER DIMENSION(*) C: const MKL_INT*	Pointer to an array of size n . The i -th element of the array contains the number of the group the observation belongs to.
<i>pld_mean</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslssreditpooledcovariance REAL(KIND=8) DIMENSION(*) for vslsreditpooledcovariance C: const float* for vslsSEditPooledCovariance const double* for vslsSEEditPooledCovariance	Pointer to the array of pooled means
<i>pld_cov</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslssreditpooledcovariance REAL(KIND=8) DIMENSION(*) for vslsreditpooledcovariance C: const float* for vslsSEEditPooledCovariance const double* for vslsSEEditPooledCovariance	Pointer to the array that holds a pooled covariance matrix
<i>req_grp_indices</i>	Fortran: INTEGER DIMENSION(*) C: const MKL_INT*	Pointer to the array that contains indices of groups for which estimates to return (such as covariance and mean)
<i>grp_means</i>	Fortran: REAL(KIND=4) DIMENSION(*) for product=Fortran vslssreditpooledcovariance REAL(KIND=8) DIMENSION(*) for vslsreditpooledcovariance C: const float* for vslsSEEditPooledCovariance const double* for vslsSEEditPooledCovariance	Pointer to the array of group means
<i>grp_cov</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslssreditpooledcovariance REAL(KIND=8) DIMENSION(*) for vslsreditpooledcovariance C: const float* for vslsSEEditPooledCovariance const double* for vslsSEEditPooledCovariance	Pointer to the array that holds group covariance matrices

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Current status of the task

Description

The `vslSSEditPooledCovariance` routine replaces pointers to the array of group indices, the array of pooled means, the array for a pooled covariance matrix, and pointers to the array of indices of group matrices, the array of group means, and the array for group covariance matrices with values passed in the editors. If an input parameter is `NULL`, the corresponding parameter in the Summary Statistics task descriptor remains unchanged. Use the `vslSSEditTask` routine to replace the storage format for pooled and group covariance matrices.

`vslSSEditRobustCovariance`

Modifies pointers to arrays related to a robust covariance matrix.

Syntax

Fortran:

```
status = vslsseditrobustcovariance(task, rcov_storage, nparams, params, rmean, rcov)
status = vslseditrobustcovariance(task, rcov_storage, nparams, params, rmean, rcov)
```

C:

```
status = vslsSSEditRobustCovariance(task, rcov_storage, nparams, params, rmean, rcov);
status = vslsSSEditRobustCovariance(task, rcov_storage, nparams, params, rmean, rcov);
```

Include Files

- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(VSL_SS_TASK) C: VSLSSTaskPtr	Descriptor of the task
<i>rcov_storage</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the storage format of a robust covariance matrix
<i>nparams</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the number of method parameters
<i>params</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditrobustcovariance REAL(KIND=8) DIMENSION(*) for	Pointer to the array of method parameters

Name	Type	Description
	<pre> vsldsseditrobustcovariance C: const float* for vslsSSEditRobustCovariance const double* for vslsSSEditRobustCovariance </pre>	
rmean	<p>Fortran: REAL(KIND=4) DIMENSION(*) for</p> <pre> vslsseditrobustcovariance REAL(KIND=8) DIMENSION(*) for vsldsseditrobustcovariance C: const float* for vslsSSEditRobustCovariance const double* for vslsSSEditRobustCovariance </pre>	Pointer to the array of robust means
rcov	<p>Fortran: REAL(KIND=4) DIMENSION(*) for</p> <pre> vslsseditrobustcovariance REAL(KIND=8) DIMENSION(*) for vsldsseditrobustcovariance C: const float* for vslsSSEditRobustCovariance const double* for vslsSSEditRobustCovariance </pre>	Pointer to a robust covariance matrix

Output Parameters

Name	Type	Description
status	<p>Fortran: INTEGER</p> <p>C: int</p>	Current status of the task

Description

The `vslSSEditRobustCovariance` routine uses values passed as parameters of the routine to replace:

- pointers to covariance matrix storage
- pointers to the number of method parameters and to the array of the method parameters of size `nparams`
- pointers to the arrays that hold robust means and covariance

See [Table "Storage formats of a variance-covariance/correlation matrix"](#) for possible values of the `rcov_storage` parameter. If an input parameter is `NULL`, the corresponding parameter in the task descriptor remains unchanged.

Intel MKL provides a Translated Biweight S-estimator (TBS) for robust estimation of a variance-covariance matrix and mean [Rocke96]. Use one iteration of the Maronna algorithm with the reweighting step [Maronna02] to compute the initial point of the algorithm. Pack the parameters of the TBS algorithm into the *params* array and pass them into the editor. Table "Structure of the Array of TBS Parameters" describes the *params* structure.

Structure of the Array of TBS Parameters

Array Position	Algorithm Parameter	Description
0	ϵ	Breakdown point, the number of outliers the algorithm can hold. By default, the value is $(n-p) / (2n)$.
1	α	Asymptotic rejection probability, see details in [Rocke96]. By default, the value is 0.001.
2	δ	Stopping criterion: the algorithm is terminated if weights are changed less than δ . By default, the value is 0.001.
3	max_iter	Maximum number of iterations. The algorithm terminates after max_iter iterations. By default, the value is 10. If you set this parameter to zero, the function returns a robust estimate of the variance-covariance matrix computed using the Maronna method [Maronna02] only.

The robust estimator of variance-covariance implementation in Intel MKL requires that the number of observations n be greater than twice the number of variables: $n > 2p$.

See additional details of the algorithm usage model in the *Intel® MKL Summary Statistics Library Application Notes* document [[SSL Notes](#)].

vsISSEditOutliersDetection

Modifies array pointers related to multivariate outliers detection.

Syntax

Fortran:

```
status = vslssreditoutliersdetection(task, nparams, params, w)
status = vsldssreditoutliersdetection(task, nparams, params, w)
```

C:

```
status = vslsSSEditOutliersDetection(task, nparams, params, w);
status = vsldSSEditOutliersDetection(task, nparams, params, w);
```

Include Files

- Fortran 90: mkl_vsl.f90
- C: mkl.h

Input Parameters

Name	Type	Description
task	Fortran: TYPE(VSL_SS_TASK) C: VSLSSTaskPtr	Descriptor of the task

Name	Type	Description
<i>nparams</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the number of method parameters
<i>params</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditoutliersdetection REAL(KIND=8) DIMENSION(*) for vsldsseditoutliersdetection C: const float* for vslsSSEditOutliersDetection const double* for vsldSSEEditOutliersDetection	Pointer to the array of method parameters
<i>w</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditoutliersdetection REAL(KIND=8) DIMENSION(*) for vsldsseditoutliersdetection C: const float* for vslsSSEditOutliersDetection const double* for vsldSSEEditOutliersDetection	Pointer to an array of size <i>n</i> . The array holds the weights of observations to be marked as outliers.

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Current status of the task

Description

The `vslSSEditOutliersDetection` routine uses the parameters passed to replace

- the pointers to the number of method parameters and to the array of the method parameters of size *nparams*
- the pointer to the array that holds the calculated weights of the observations

If an input parameter is `NULL`, the corresponding parameter in the task descriptor remains unchanged.

Intel MKL provides the BACON algorithm ([Billor00]) for the detection of multivariate outliers. Pack the parameters of the BACON algorithm into the *params* array and pass them into the editor. Table "Structure of the Array of BACON Parameters" describes the *params* structure.

Structure of the Array of BACON Parameters

Array Position	Algorithm Parameter	Description
0	Method to start the algorithm	The parameter takes one of the following possible values: <code>VSL_SS_METHOD_BACON_MEDIAN_INIT</code> , if the algorithm is started using the median estimate. This is the default value of the parameter.

Array Position	Algorithm Parameter	Description
		VSL_SS_METHOD_BACON_MAHALANOBIS_INIT, if the algorithm is started using the Mahalanobis distances.
1	α	One-tailed probability that defines the $(1 - \alpha)$ quantile of χ^2 distribution with p degrees of freedom. The recommended value is α / n , where n is the number of observations. By default, the value is 0.05.
2	δ	Stopping criterion; the algorithm is terminated if the size of the basic subset is changed less than δ . By default, the value is 0.005.

Output of the algorithm is the vector of weights, `BaconWeights`, such that `BaconWeights(i) = 0` if i -th observation is detected as an outlier. Otherwise `BaconWeights(i) = w(i)`, where `w` is the vector of input weights. If you do not provide the vector of input weights, `BaconWeights(i)` is set to 1 if the i -th observation is not detected as an outlier.

See additional details about usage model of the algorithm in the *Intel(R) MKL Summary Statistics Library Application Notes* document [[SSL Notes](#)].

vslSSEditMissingValues

Modifies pointers to arrays associated with the method of supporting missing values in a dataset.

Syntax

Fortran:

```
status = vslssreditmissingvalues(task, nparams, params, init_estimates_n,
init_estimates, prior_n, prior, simul_missing_vals_n, simul_missing_vals, estimates_n,
estimates)
```

```
status = vsldssreditmissingvalues(task, nparams, params, init_estimates_n,
init_estimates, prior_n, prior, simul_missing_vals_n, simul_missing_vals, estimates_n,
estimates)
```

C:

```
status = vslsSSEditMissingValues(task, nparams, params, init_estimates_n,
init_estimates, prior_n, prior, simul_missing_vals_n, simul_missing_vals, estimates_n,
estimates);
```

```
status = vsldSSEditMissingValues(task, nparams, params, init_estimates_n,
init_estimates, prior_n, prior, simul_missing_vals_n, simul_missing_vals, estimates_n,
estimates);
```

Include Files

- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	Fortran: TYPE(VSL_SS_TASK)	Descriptor of the task

Name	Type	Description
<i>nparams</i>	C: VSLSSSTaskPtr Fortran: INTEGER C: const MKL_INT*	Pointer to the number of method parameters
<i>params</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslssseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues C: const float* for vslsSSEditMissingValues const double* for vsldSSEditMissingValues	Pointer to the array of method parameters
<i>init_estimates_n</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the number of initial estimates for mean and a variance-covariance matrix
<i>init_estimates</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslssseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues C: const float* for vslsSSEditMissingValues const double* for vsldSSEditMissingValues	Pointer to the array that holds initial estimates for mean and a variance-covariance matrix
<i>prior_n</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the number of prior parameters
<i>prior</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslssseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues C: const float* for vslsSSEditMissingValues const double* for vsldSSEditMissingValues	Pointer to the array of prior parameters
<i>simul_missing_vals_n</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the size of the array that holds output of the Multiple Imputation method
<i>simul_missing_vals</i>	Fortran: REAL(KIND=4) DIMENSION(*) for vslssseditmissingvalues REAL(KIND=8) DIMENSION(*) for vsldsseditmissingvalues	Pointer to the array of size $k*m$, where k is the total number of missing values, and m is number of copies of missing values. The array

Name	Type	Description
	C: const float* for vslssSEditMissingValues const double* for vslsSEditMissingValues	holds m sets of simulated missing values for the matrix of observations.
estimates_n	Fortran: INTEGER C: const MKL_INT*	Pointer to the number of estimates to be returned by the routine
estimates	Fortran: REAL(KIND=4) DIMENSION(*) for vslsseditmissingvalues REAL(KIND=8) DIMENSION(*) for vslseditmissingvalues C: const float* for vslssSEditMissingValues const double* for vslsSEditMissingValues	Pointer to the array that holds estimates of the mean and a variance-covariance matrix.

Output Parameters

Name	Type	Description
status	Fortran: INTEGER C: int	Current status of the task

Description

The `vslssSEditMissingValues` routine uses values passed as parameters of the routine to replace pointers to the number and the array of the method parameters, pointers to the number and the array of initial mean/variance-covariance estimates, the pointer to the number and the array of prior parameters, pointers to the number and the array of simulated missing values, and pointers to the number and the array of the intermediate mean/covariance estimates. If an input parameter is `NULL`, the corresponding parameter in the task descriptor remains unchanged.

Before you call the Summary Statistics routines to process missing values, preprocess the dataset and denote missing observations with one of the following predefined constants:

- `VSL_SS_SNAN`, if the dataset is stored in single precision floating-point arithmetic
- `VSL_SS_DNAN`, if the dataset is stored in double precision floating-point arithmetic

Intel MKL provides the `VSL_SS_METHOD_MI` method to support missing values in the dataset based on the Multiple Imputation (MI) approach described in [Schafer97]. The following components support Multiple Imputation:

- Expectation Maximization (EM) algorithm to compute the start point for the Data Augmentation (DA) procedure
- DA function

NOTE

The DA component of the MI procedure is simulation-based and uses the `VSL_BRNG_MCG59` basic random number generator with predefined `seed = 250` and the Gaussian distribution generator (`ICDF` method) available in Intel MKL [Gaussian].

Pack the parameters of the MI algorithm into the `params` array. [Table "Structure of the Array of MI Parameters"](#) describes the `params` structure.

Structure of the Array of MI Parameters

Array Position	Algorithm Parameter	Description
0	<code>em_iter_num</code>	Maximal number of iterations for the EM algorithm. By default, this value is 50.
1	<code>da_iter_num</code>	Maximal number of iterations for the DA algorithm. By default, this value is 30.
2	ε	Stopping criterion for the EM algorithm. The algorithm terminates if the maximal module of the element-wise difference between the previous and current parameter values is less than ε . By default, this value is 0.001.
3	<code>m</code>	Number of sets to impute
4	<code>missing_vals_num</code>	Total number of missing values in the datasets

You can also pass initial estimates into the EM algorithm by packing both the vector of means and the variance-covariance matrix as a one-dimensional array `init_estimates`. The size of the array should be at least $p + p(p + 1)/2$. For $i=0, \dots, p-1$, the `init_estimates[i]` array contains the initial estimate of means. The remaining positions of the array are occupied by the upper triangular part of the variance-covariance matrix.

If you provide no initial estimates for the EM algorithm, the editor uses the default values, that is, the vector of zero means and the unitary matrix as a variance-covariance matrix. You can also pass `prior` parameters for μ and Σ into the library: μ_0 , τ , m , and Λ^{-1} . Pack these parameters as a one-dimensional array `prior` with a size of at least

$$(p^2 + 3p + 4)/2.$$

The storage format is as follows:

- `prior[0], \dots, prior[p-1]` contain the elements of the vector μ_0 .
- `prior[p]` contains the parameter τ .
- `prior[p+1]` contains the parameter m .
- The remaining positions are occupied by the upper-triangular part of the inverted matrix Λ^{-1} .

If you provide no `prior` parameters, the editor uses their default values:

- The array of p zeros is used as μ_0 .
- τ is set to 0.
- m is set to p .
- The zero matrix is used as an initial approximate of Λ^{-1} .

The `EditMissingValues` editor returns m sets of imputed values and/or a sequence of parameter estimates drawn during the DA procedure.

The editor returns the imputed values as the `simul_missing_vals` array. The size of the array should be sufficient to hold m sets each of the `missing_vals_num` size, that is, at least $m * missing_vals_num$ in total. The editor packs the imputed values one by one in the order of their appearance in the matrix of observations.

For example, consider a task of dimension 4. The total number of observations n is 10. The second observation vector misses variables 1 and 2, and the seventh observation vector lacks variable 1. The number of sets to impute is $m=2$. Then, `simul_missing_vals[0]` and `simul_missing_vals[1]` contains the first and the second points for the second observation vector, and `simul_missing_vals[2]` holds the first point for the seventh observation. Positions 3, 4, and 5 are formed similarly.

To estimate convergence of the DA algorithm and choose a proper value of the number of DA iterations, request the sequence of parameter estimates that are produced during the DA procedure. The editor returns the sequence of parameters as a single array. The size of the array is

$$m \cdot da_iter_num \cdot (p + (p^2 + p) / 2)$$

where

- m is the number of sets of values to impute.
- `da_iter_num` is the number of DA iterations.
- The value $p + (p^2 + p) / 2$ determines the size of the memory to hold one set of the parameter estimates.

In each set of the parameters, the vector of means occupies the first p positions and the remaining $(p^2 + p) / 2$ positions are intended for the upper triangular part of the variance-covariance matrix.

Upon successful generation of m sets of imputed values, you can place them in cells of the data matrix with missing values and use the Summary Statistics routines to analyze and get estimates for each of the m complete datasets.

NOTE

Intel MKL implementation of the MI algorithm rewrites cells of the dataset that contain the `VSL_SS_SNAN`/`VSL_SS_DNAN` values. If you want to use the Summary Statistics routines to process the data with missing values again, mask the positions of the empty cells.

See additional details of the algorithm usage model in the *Intel® MKL Summary Statistics Library Application Notes* document [[SSL Notes](#)].

[vslSSEditCorParameterization](#)

Modifies pointers to arrays related to the algorithm of correlation matrix parameterization.

Syntax

Fortran:

```
status = vslsseditcorparameterization(task, cor, cor_storage, pcor, pcor_storage)
status = vslseditcorparameterization(task, cor, cor_storage, pcor, pcor_storage)
```

C:

```
status = vslsSEditCorParameterization(task, cor, cor_storage, pcor, pcor_storage);
status = vslsSEditCorParameterization(task, cor, cor_storage, pcor, pcor_storage);
```

Include Files

- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE (VSL_SS_TASK) C: VSLSSTaskPtr	Descriptor of the task
<i>cor</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vslsseditcorparameterization REAL (KIND=8) DIMENSION(*) for vsldsseditcorparameterization C: const float* for vslsSSEditCorParameterization const double* for vsldSSEditCorParameterization	Pointer to the correlation matrix
<i>cor_storage</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the storage format of the correlation matrix
<i>pcor</i>	Fortran: REAL (KIND=4) DIMENSION(*) for vslsseditcorparameterization REAL (KIND=8) DIMENSION(*) for vsldsseditcorparameterization C: const float* for vslsSSEditCorParameterization const double* for vsldSSEditCorParameterization	Pointer to the parameterized correlation matrix
<i>por_storage</i>	Fortran: INTEGER C: const MKL_INT*	Pointer to the storage format of the parameterized correlation matrix

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Current status of the task

Description

The `vslSSEditCorParameterization` routine uses values passed as parameters of the routine to replace pointers to the correlation matrix, pointers to the correlation matrix storage format, a pointer to the parameterized correlation matrix, and a pointer to the parameterized correlation matrix storage format. See [Table "Storage formats of a variance-covariance/correlation matrix"](#) for possible values of the `cor_storage` and `pcor_storage` parameters. If an input parameter is NULL, the corresponding parameter in the Summary Statistics task descriptor remains unchanged.

Task Computation Routines

Task computation routines calculate statistical estimates on the data provided and parameters held in the task descriptor. After you create the task and initialize its parameters, you can call the computation routines as many times as necessary. [Table "Summary Statistics Estimates Obtained with vsISSCompute Routine"](#) lists the statistical estimates that you can obtain using the `vsISSCompute` routine.

NOTE

The Summary Statistics computation routines do not signal floating-point errors, such as overflow or gradual underflow, or operations with NaNs (except for the missing values in the observations).

Summary Statistics Estimates Obtained with vsISSCompute Routine

Estimate	Support of Observations Available in Blocks	Description
<code>VSL_SS_MEAN</code>	Yes	Computes the array of means.
<code>VSL_SS_SUM</code>	Yes	Computes the array of sums.
<code>VSL_SS_2R_MOM</code>	Yes	Computes the array of the 2 nd order raw moments.
<code>VSL_SS_2R_SUM</code>	Yes	Computes the array of raw sums of the 2 nd order.
<code>VSL_SS_3R_MOM</code>	Yes	Computes the array of the 3 rd order raw moments.
<code>VSL_SS_3R_SUM</code>	Yes	Computes the array of raw sums of the 3 rd order.
<code>VSL_SS_4R_MOM</code>	Yes	Computes the array of the 4 th order raw moments.
<code>VSL_SS_4R_SUM</code>	Yes	Computes the array of raw sums of the 4 th order.
<code>VSL_SS_2C_MOM</code>	Yes	Computes the array of the 2 nd order central moments.
<code>VSL_SS_2C_SUM</code>	Yes	Computes the array of central sums of the 2 nd order.
<code>VSL_SS_3C_MOM</code>	Yes	Computes the array of the 3 rd order central moments.
<code>VSL_SS_3C_SUM</code>	Yes	Computes the array of central sums of the 3 rd order.
<code>VSL_SS_4C_MOM</code>	Yes	Computes the array of the 4 th order central moments.
<code>VSL_SS_4C_SUM</code>	Yes	Computes the array of central sums of the 4 th order.
<code>VSL_SS_KURTOSIS</code>	Yes	Computes the array of kurtosis values.
<code>VSL_SS_SKEWNESS</code>	Yes	Computes the array of skewness values.
<code>VSL_SS_MIN</code>	Yes	Computes the array of minimum values.

Estimate	Support of Observations Available in Blocks	Description
VSL_SS_MAX	Yes	Computes the array of maximum values.
VSL_SS_VARIATION	Yes	Computes the array of variation coefficients.
VSL_SS_COV	Yes	Computes a covariance matrix.
VSL_SS_COR	Yes	Computes a correlation matrix. The main diagonal of the correlation matrix holds variances of the random vector components.
VSL_SS_CP	Yes	Computes a cross-product matrix.
VSL_SS_POOLED_COV	No	Computes a pooled covariance matrix.
VSL_SS_POOLED_MEAN	No	Computes an array of pooled means.
VSL_SS_GROUP_COV	No	Computes group covariance matrices.
VSL_SS_GROUP_MEAN	No	Computes group means.
VSL_SS_QUANTS	No	Computes quantiles.
VSL_SS_ORDER_STATS	No	Computes order statistics.
VSL_SS_ROBUST_COV	No	Computes a robust covariance matrix.
VSL_SS_OUTLIERS	No	Detects outliers in the dataset.
VSL_SS_PARTIAL_COV	No	Computes a partial covariance matrix.
VSL_SS_PARTIAL_COR	No	Computes a partial correlation matrix.
VSL_SS_MISSING_VALS	No	Supports missing values in datasets.
VSL_SS_PARAMTR_COR	No	Computes a parameterized correlation matrix.
VSL_SS_STREAM_QUANTS	Yes	Computes quantiles for streaming data.
VSL_SS_MDAD	No	Computes median absolute deviation.
VSL_SS_MNAD	No	Computes mean absolute deviation.

[Table "Summary Statistics Computation Method"](#) lists estimate calculation methods supported by Intel MKL. See the *Intel(R) MKL Summary Statistics Library Application Notes* document [[SSL Notes](#)] for a detailed description of the methods.

Summary Statistics Computation Method

Method	Description
VSL_SS_METHOD_FAST	<p>Fast method for calculation of the estimates:</p> <ul style="list-style-type: none"> • raw/central moments/sums, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix • min/max/quantile/order statistics • partial variance-covariance • median/mean absolute deviation
VSL_SS_METHOD_FAST_USER_MEAN	Fast method for calculation of the estimates given user-defined mean:

Method	Description
	<ul style="list-style-type: none"> central moments/sums of 2-4 order, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix, mean absolute deviation
VSL_SS_METHOD_1PASS	One-pass method for calculation of estimates:
	<ul style="list-style-type: none"> raw/central moments/sums, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix pooled/group covariance matrix
VSL_SS_METHOD_TBS	TBS method for robust estimation of covariance and mean
VSL_SS_METHOD_BACON	BACON method for detection of multivariate outliers
VSL_SS_METHOD_MI	Multiple imputation method for support of missing values
VSL_SS_METHOD_SD	Spectral decomposition method for parameterization of a correlation matrix
VSL_SS_METHOD_SQUANTS_ZW	Zhang-Wang (ZW) method for quantile estimation for streaming data
VSL_SS_METHOD_SQUANTS_ZW_FAST	Fast ZW method for quantile estimation for streaming data

You can calculate all requested estimates in one call of the routine. For example, to compute a kurtosis and covariance matrix using a fast method, pass a combination of the pre-defined parameters into the Compute routine as shown in the example below:

```
...
method = VSL_SS_METHOD_FAST;
task_params = VSL_SS_KURTOSIS|VSL_SS_COV;
...
status = vsldSSCompute( task, task_params, method );
```

To compute statistical estimates for the next block of observations, you can do one of the following:

- copy the observations to memory, starting with the address available to the task
- use one of the appropriate [Editors](#) to modify the pointer to the new dataset in the task.

The library does not detect your changes of the dataset and computed statistical estimates. To obtain statistical estimates for a new matrix, change the observations and initialize relevant arrays. You can follow this procedure to compute statistical estimates for observations that come in portions. See [Table "Summary Statistics Estimates Obtained with vs1SSCompute Routine"](#) for information on such observations supported by the Intel MKL Summary Statistics estimators.

To modify parameters of the task using the Task Editors, set the address of the targeted matrix of the observations or change the respective vector component indices. After you complete editing the task parameters, you can compute statistical estimates in the modified environment.

If the task completes successfully, the computation routine returns the zero status code. If an error is detected, the computation routine returns an error code. In particular, an error status code is returned in the following cases:

- the task pointer is `NULL`
- memory allocation has failed
- the calculation has failed for some other reason

NOTE

You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

vslSSCompute

Computes Summary Statistics estimates.

Syntax**Fortran:**

```
status = vslssscompute(task, estimates, method)
status = vslldsscompute(task, estimates, method)
```

C:

```
status = vsllsSSCompute(task, estimates, method);
status = vsldSSCompute(task, estimates, method);
```

Include Files

- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(VSL_SS_TASK) C: VSLSSTaskPtr	Descriptor of the task
<i>estimates</i>	Fortran: INTEGER (KIND=8) C: const MKL_INT64	List of statistical estimates to compute
<i>method</i>	Fortran: INTEGER C: const MKL_INT	Method to be used in calculations

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Current status of the task

Description

The `vslSSCompute` routine calculates statistical estimates passed as the `estimates` parameter using the algorithms passed as the `method` parameter of the routine. The computations are done in the context of the task descriptor that contains pointers to all required and optional, if necessary, properly initialized arrays. In one call of the function, you can compute several estimates using proper methods for their calculation. See [Table "Summary Statistics Estimates Obtained with Compute Routine"](#) for the list of the estimates that you can calculate with the `vslSSCompute` routine. See [Table "Summary Statistics Computation Methods"](#) for the list of possible values of the `method` parameter.

To initialize single or double precision version task parameters, use the single (`vslssscompute`) or double (`vslldsscompute`) version of the editor, respectively. To initialize parameters of the integer type, use an integer version of the editor (`vslisscompute`).

NOTE

Requesting a combination of the `VSL_SS_MISSING_VALS` value and any other estimate parameter in the `Compute` function results in processing only the missing values.

Application Notes

Be aware that when computing a correlation matrix, the `vslSSCompute` routine allocates an additional array for each thread which is running the task. If you are running on a large number of threads `vslSSCompute` might consume large amounts of memory.

When calculating covariance, correlation, or cross product, the number of bytes of memory required is at least $(P*P*T + P*T)*b$, where P is the dimension of the task or number of variables, T is the number of threads, and b is the number of bytes required for each unit of data. If observation is weighted and the method is `VSL_SS_METHOD_FAST`, then the memory required is at least $(P*P*T + P*T + N*P)*b$, where N is the number of observations.

Task Destructor

Task destructor is the `vslSSDeleteTask` routine intended to delete task objects and release memory.

vslSSDeleteTask

Destroys the task object and releases the memory.

Syntax

Fortran:

```
status = vslssdeletetask(task)
```

C:

```
status = vslSSDeleteTask(&task);
```

Include Files

- Fortran 90: `mkl_vsl.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	Fortran: <code>TYPE(VSL_SS_TASK)</code> C: <code>VSLSSTaskPtr*</code>	Descriptor of the task to destroy

Output Parameters

Name	Type	Description
<code>status</code>	Fortran: <code>INTEGER</code> C: <code>int</code>	Sets to <code>VSL_STATUS_OK</code> if the task is deleted; otherwise a non-zero code is returned.

Description

The `vslSSDeleteTask` routine deletes the task descriptor object, releases the memory allocated for the structure, and sets the task pointer to `NULL`. If `vslSSDeleteTask` fails to delete the task successfully, it returns an error code.

NOTE

Call of the destructor with the `NULL` pointer as the parameter results in termination of the function with no system crash.

Usage Examples

The following examples show various standard operations with Summary Statistics routines.

Calculating Fixed Estimates for Fixed Data

The example shows recurrent calculation of the same estimates with a given set of variables for the complete life cycle of the task in the case of a variance-covariance matrix. The set of vector components to process remains unchanged, and the data comes in blocks. Before you call the `vslSSCompute` routine, initialize pointers to arrays for mean and covariance and set buffers.

```
...
double w[2];
double indices[DIM] = {1, 0, 1};

/* calculating mean for 1st and 3d random vector components */

/* Initialize parameters of the task */
p = DIM;
n = N;

xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
covstorage = VSL_SS_MATRIX_STORAGE_FULL;

w[0] = 0.0; w[1] = 0.0;

for ( i = 0; i < p; i++ ) mean[i] = 0.0;
for ( i = 0; i < p*p; i++ ) cov[i] = 0.0;

status = vsldSSNewTask( &task, &p, &n, &xstorage, x, 0, indices );

status = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, w );
status = vsldSSEditCovCor( task, mean, cov, &covstorage, 0, 0 );
```

You can process data arrays that come in blocks as follows:

```
for ( i = 0; i < num_of_blocks; i++ )
{
    status = vsldSSCompute( task, VSL_SS_COV, VSL_SS_METHOD_FAST );
    /* Read new data block into array x */
}
```

Calculating Different Estimates for Variable Data

The context of your calculation may change in the process of data analysis. The example below shows the data that comes in two blocks. You need to estimate a covariance matrix for the complete data, and the third central moment for the second block of the data using the weights that were accumulated for the previous datasets. The second block of the data is stored in another array. You can proceed as follows:

```

/* Set parameters for the task */
p = DIM;
n = N;
xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
covstorage = VSL_SS_MATRIX_STORAGE_FULL;

w[0] = 0.0; w[1] = 0.0;

for ( i = 0; i < p; i++ ) mean[i] = 0.0;
for ( i = 0; i < p*p; i++ ) cov[i] = 0.0;

/* Create task */
status = vslSSNewTask( &task, &p, &n, &xstorage, x1, 0, indices );

/* Initialize the task parameters */
status = vslSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, w );
status = vslSSEditCovCor( task, mean, cov, &covstorage, 0, 0 );

/* Calculate covariance for the x1 data */
status = vslSSCompute( task, VSL_SS_COV, VSL_SS_METHOD_FAST );

/* Initialize array of the 3d central moments and pass the pointer to the task */
for ( i = 0; i < p; i++ ) c3_m[i] = 0.0;

/* Modify task context */
status = vslSSEditTask( task, VSL_SS_ED_3C_MOM, c3_m );
status = vslSSEditTask( task, VSL_SS_ED_OBSERV, x2 );

/* Calculate covariance for the x1 & x2 data block */
/* Calculate the 3d central moment for the 2nd data block using earlier accumulated weight */
status = vslSSCompute(task, VSL_SS_COV|VSL_SS_3C_MOM, VSL_SS_METHOD_FAST );
...
status = vslSSDeleteTask( &task );

```

Similarly, you can modify indices of the variables to be processed for the next data block.

Mathematical Notation and Definitions

The following notations are used in the mathematical definitions and the description of the Intel MKL Summary Statistics functions.

Matrix and Weights of Observations

For a random p -dimensional vector $\xi = (\xi_1, \dots, \xi_i, \dots, \xi_p)$, this manual denotes the following:

- $(X)_{i=(x_{ij})_{j=1..n}}$ is the result of n independent observations for the i -th component ξ_i of the vector ξ .
- The two-dimensional array $X=(x_{ij})_{p \times n}$ is the matrix of observations.
- The column $[X]_{j=(x_{ij})_{i=1..p}}$ of the matrix X is the j -th observation of the random vector ξ .

Each observation $[X]_j$ is assigned a non-negative weight w_j , where

- The vector $(w_j)_{j=1..n}$ is a vector of weights corresponding to n observations of the random vector ξ .
-

$$W = \sum_{i=1}^n w_i$$

is the accumulated weight corresponding to observations X .

Vector of sample means

$$M(X) = (M_1(X), \dots, M_p(X)) \text{ with } M_i(X) = \frac{1}{W} \sum_{j=1}^n w_j x_{ij}$$

for all $i = 1, \dots, p$.

Vector of sample partial sums

$$S(X) = (S_1(X), \dots, S_p(X)) \text{ with } S_i(X) = \sum_{j=1}^n w_j x_{ij}$$

for all $i = 1, \dots, p$.

Vector of sample variances

$$V(X) = (V_1(X), \dots, V_p(X)) \text{ with } V_i(X) = \frac{1}{B} \sum_{j=1}^n w_j (x_{ij} - M_i(X))^2, B = W - \sum_{j=1}^n w_j^2 / W$$

for all $i = 1, \dots, p$.

Vector of sample raw/algebraic moments of k -th order, $k \geq 1$

$$R^{(k)}(X) = (R_1^{(k)}(X), \dots, R_p^{(k)}(X)) \text{ with } R_i^{(k)}(X) = \frac{1}{W} \sum_{j=1}^n w_j x_{ij}^k$$

for all $i = 1, \dots, p$.

Vector of sample raw/algebraic partial sums of k -th order, $k = 2, 3, 4$ (raw/algebraic partial sums of squares/cubes/fourth powers)

$$S^k(X) = (S_1^k(X), \dots, S_p^k(X)) \text{ with } S_i^k(X) = \sum_{j=1}^n w_j x_{ij}^k$$

for all $i = 1, \dots, p$.

Vector of sample central moments of the third and the fourth order

$$C^{(k)}(X) = (C_1^{(k)}(X), \dots, C_p^{(k)}(X)) \text{ with } C_i^{(k)}(X) = \frac{1}{B} \sum_{j=1}^n w_j (x_{ij} - M_i(X))^k, B = \sum_{j=1}^n w_j$$

for all $i = 1, \dots, p$ and $k = 3, 4$.

Vector of sample central partial sums of k -th order, $k = 2, 3, 4$ (central partial sums of squares/cubes/fourth powers)

$$S^k(X) = (S_1^k(X), \dots, S_p^k(X)) \text{ with } S_i^k(X) = \sum_{j=1}^n w_j (x_{ij} - S_i(X))^k$$

for all $i = 1, \dots, p$.

Vector of sample excess kurtosis values

$$B(X) = (B_1(X), \dots, B_p(X)) \text{ with } B_i(X) = \frac{C_i^{(4)}(X)}{V_i^2(X)} - 3$$

for all $i = 1, \dots, p$.

Vector of sample skewness values

$$\Gamma(X) = (\Gamma_1(X), \dots, \Gamma_p(X)) \text{ with } \Gamma_i(X) = \frac{C_i^{(3)}(X)}{V_i^{1.5}(X)}$$

for all $i = 1, \dots, p$.

Vector of sample variation coefficients

$$VC(X) = (VC_1(X), \dots, VC_p(X)) \text{ with } VC_i(X) = \frac{V_i^{0.5}(X)}{M_i(X)}$$

for all $i = 1, \dots, p$.

Matrix of order statistics

Matrix $Y = (Y_{ij})_{pxn}$, in which the i -th row $(Y)_i = (Y_{ij})_{j=1\dots n}$ is obtained as a result of sorting in the ascending order of row $(X)_i = (x_{ij})_{j=1\dots n}$ in the original matrix of observations.

Vector of sample minimum values

$$Min(X) = (Min_1(X), \dots, Min_p(X)), \text{ where } Min_i(X) = y_{i1}$$

for all $i = 1, \dots, p$.

Vector of sample maximum values

$$\text{Max}(X) = (\text{Max}_1(X), \dots, \text{Max}_p(X)), \text{ where } \text{Max}_i(X) = y_{i\text{m}}$$

for all $i = 1, \dots, p$.

Vector of sample median values

$$\text{Med}(X) = (\text{Med}_1(X), \dots, \text{Med}_p(X)), \text{ where } \text{Med}_i(X) = \begin{cases} y_{i,(n+1)/2}, & \text{if } n \text{ is odd} \\ (y_{i,n/2} + y_{i,n/2+1})/2, & \text{if } n \text{ is even} \end{cases}$$

for all $i = 1, \dots, p$.

Vector of sample median absolute deviations

$$MDAD(X) = (MDAD_1(X), \dots, MDAD_p(X)) \text{ where}$$

$$MDAD_i(X) = \text{Med}_i(Z) \text{ with } Z = (z_{ij})_{i=1 \dots p, j=1 \dots n}, z_{ij} = |x_{ij} - \text{Med}_i(X)|$$

for all $i = 1, \dots, p$.

Vector of sample mean absolute deviations

$$\dots, (|x_{11} - \bar{x}|, \dots, |x_{1n} - \bar{x}|), (|x_{21} - \bar{x}|, \dots, |x_{2n} - \bar{x}|), \dots, (|x_{p1} - \bar{x}|, \dots, |x_{pn} - \bar{x}|)$$

$$\dots, (|x_{11} - \bar{x}|, \dots, |x_{1n} - \bar{x}|), (|x_{21} - \bar{x}|, \dots, |x_{2n} - \bar{x}|), \dots, (|x_{p1} - \bar{x}|, \dots, |x_{pn} - \bar{x}|),$$

for all $i = 1, \dots, p$.

Vector of sample quantile values

For a positive integer number q and k belonging to the interval $[0, q-1]$, point z_i is the k -th q quantile of the random variable ξ_i if $P\{\xi_i \leq z_i\} \geq \beta$ and $P\{\xi_i \leq z_i\} \geq 1 - \beta$, where

- P is the probability measure.
- $\beta = k/n$ is the quantile order.

The calculation of quantiles is as follows:

$j = [(n-1)\beta]$ and $f = \{(n-1)\beta\}$ as integer and fractional parts of the number $(n-1)\beta$, respectively, and the vector of sample quantile values is

$$Q(X, \beta) = (Q_1(X, \beta), \dots, Q_p(X, \beta))$$

where

$$(Q_i(X, \beta)) = y_{i,j+1} + f(y_{i,j+2} - y_{i,j+1})$$

for all $i = 1, \dots, p$.

Variance-covariance matrix

$$C(X) = (c_{ij}(X))_{p \times p}$$

where

$$c_{ij}(X) = \frac{1}{B} \sum_{k=1}^n w_k (x_{ik} - M_i(X))(x_{jk} - M_j(X)), B = W - \sum_{j=1}^n w_j^2 / W$$

Cross-product matrix (matrix of cross-products and sums of squares)

$$CP(X) = (cp_{ij}(X))_{p \times p}$$

where

$$cp_{ij}(X) = \sum_{k=1}^n w_k (x_{ik} - M_i(X))(x_{jk} - M_j(X))$$

Pooled and group variance-covariance matrices

The set $N = \{1, \dots, n\}$ is partitioned into non-intersecting subsets

$$G_i, i = 1..g, N = \bigcup_{i=1}^g G_i$$

The observation $[X]_j = (x_{ij})_{i=1..p}$ belongs to the group r if $j \in G_r$. One observation belongs to one group only. The group mean and variance-covariance matrices are calculated similarly to the formulas above:

$$M^{(r)}(X) = (M_1^{(r)}(X), \dots, M_p^{(r)}(X)) \text{ with } M_i^{(r)}(X) = \frac{1}{W^{(r)}} \sum_{j \in G_r} w_j x_{ij}, W^{(r)} = \sum_{j \in G_r} w_j$$

for all $i = 1, \dots, p$,

$$C^{(r)}(X) = (c_{ij}^{(r)}(X))_{p \times p}$$

where

$$c_{ij}^{(r)}(X) = \frac{1}{B^{(r)}} \sum_{k \in G_r} w_k (x_{ik} - M_i^{(r)}(X))(x_{jk} - M_j^{(r)}(X)), B^{(r)} = W^{(r)} - \sum_{j \in G_r} w_j^2 / W^{(r)}$$

for all $i = 1, \dots, p$ and $j = 1, \dots, p$.

A pooled variance-covariance matrix and a pooled mean are computed as weighted mean over group covariance matrices and group means, correspondingly:

$$M^{pooled}(X) = (M_1^{pooled}(X), \dots, M_p^{pooled}(X)) \text{ with } M_i^{pooled}(X) = \frac{1}{W^{(1)} + \dots + W^{(g)}} \sum_{r=1}^g W^{(r)} M_i^{(r)}(X)$$

for all $i = 1, \dots, p$,

$$C^{pooled}(X) = (C_{ij}^{pooled}(X))_{p \times p}, C_{ij}^{pooled}(X) = \frac{1}{B^{(1)} + \dots + B^{(g)}} \sum_{r=1}^g B^{(r)} C_{ij}^{(r)}(X)$$

for all $i = 1, \dots, p$ and $j = 1, \dots, p$.

Correlation matrix

$$R(X) = (r_{ij}(X))_{p \times p}, \text{ where } r_{ij}(X) = \frac{C_{ij}}{\sqrt{C_{ii} C_{jj}}}$$

for all $i = 1, \dots, p$ and $j = 1, \dots, p$.

Partial variance-covariance matrix

For a random vector ξ partitioned into two components Z and Y , a variance-covariance matrix C describes the structure of dependencies in the vector ξ :

$$C(X) = \begin{bmatrix} C_Z(X) & C_{ZY}(X) \\ C_{YZ}(X) & C_Y(X) \end{bmatrix}.$$

The partial covariance matrix $P(X) = (p_{ij}(X))_{k \times k}$ is defined as

$$P(X) = C_Y(X) - C_{YZ}(X)C_Z^{-1}(X)C_{ZY}(X).$$

where k is the dimension of Y .

Partial correlation matrix

The following is a partial correlation matrix for all $i = 1, \dots, k$ and $j = 1, \dots, k$:

$$RP(X) = (rp_{ij}(X))_{k \times k}, \text{ where } rp_{ij}(X) = \frac{p_{ij}(X)}{\sqrt{p_{ii}(X)p_{jj}(X)}}$$

where

- k is the dimension of Y .
- $p_{ij}(X)$ are elements of the partial variance-covariance matrix.

Fourier Transform Functions

The general form of the discrete Fourier transform is

$$Z_{k_1, k_2, \dots, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \dots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1, j_2, \dots, j_d} \exp \left(\delta i 2\pi \sum_{l=1}^d j_l k_l / n_l \right)$$

for $k_1 = 0, \dots, n_1-1$ ($l = 1, \dots, d$), where σ is a scale factor, $\delta = -1$ for the forward transform, and $\delta = +1$ for the inverse (backward) transform. In the forward transform, the input (periodic) sequence $\{w_{j_1, j_2, \dots, j_d}\}$ belongs to the set of complex-valued sequences and real-valued sequences. Respective domains for the backward transform are represented by complex-valued sequences and complex-valued conjugate-even sequences.

The Intel® Math Kernel Library (Intel® MKL) provides an interface for computing a discrete Fourier transform through the fast Fourier transform algorithm. Prefixes `Dfti` in function names and `DFTI` in the names of configuration parameters stand for Discrete Fourier Transform Interface.

This chapter describes the following implementations of the fast Fourier transform functions available in Intel MKL:

- Fast Fourier transform (FFT) functions for single-processor or shared-memory systems (see [FFT Functions](#))
- [Cluster FFT functions](#) for distributed-memory architectures (available with Intel® MKL for the Linux* and Windows* operating systems only)

NOTE

Intel MKL also supports the FFTW3* interfaces to the fast Fourier transform functionality for shared memory paradigm (SMP) systems.

Both FFT and Cluster FFT functions compute an FFT in five steps:

1. Allocate a fresh descriptor for the problem with a call to the `DftiCreateDescriptor` or `DftiCreateDescriptorDM` function. The descriptor captures the configuration of the transform, such as the dimensionality (or rank), sizes, number of transforms, memory layout of the input/output data (defined by strides), and scaling factors. Many of the configuration settings are assigned default values in this call which you might need to modify in your application.
2. Optionally adjust the descriptor configuration with a call to the `DftiSetValue` or `DftiSetValueDM` function as needed. Typically, you must carefully define the data storage layout for an FFT or the data distribution among processes for a Cluster FFT. The configuration settings of the descriptor, such as the default values, can be obtained with the `DftiGetValue` or `DftiGetValueDM` function.
3. Commit the descriptor with a call to the `DftiCommitDescriptor` or `DftiCommitDescriptorDM` function, that is, make the descriptor ready for the transform computation. Once the descriptor is committed, the parameters of the transform, such as the type and number of transforms, strides and distances, the type and storage layout of the data, and so on, are "frozen" in the descriptor.
4. Compute the transform with a call to the `DftiComputeForward/DftiComputeBackward` or `DftiComputeForwardDM/DftiComputeBackwardDM` functions as many times as needed. Because the descriptor is defined and committed separately, all that the compute functions do is take the input and output data and compute the transform as defined. To modify any configuration parameters for another call to a compute function, use `DftiSetValue` followed by `DftiCommitDescriptor` (`DftiSetValueDM` followed by `DftiCommitDescriptorDM`) or create and commit another descriptor.

5. Deallocate the descriptor with a call to the [DftiFreeDescriptor](#) or [DftiFreeDescriptorDM](#) function. This returns the memory internally consumed by the descriptor to the operating system.

All the above functions return an integer status value, which is zero upon successful completion of the operation. You can interpret a non-zero status with the help of the [DftiErrorClass](#) or [DftiErrorMessage](#) function.

The FFT functions support lengths with arbitrary factors. You can improve performance of the Intel MKL FFT if the length of your data vector permits factorization into powers of optimized radices. See the *Intel MKL User's Guide* for specific radices supported efficiently.

NOTE

The FFT functions assume the Cartesian representation of complex data (that is, the real and imaginary parts define a complex number). The Intel MKL Vector Mathematical Functions provide efficient tools for conversion to and from polar representation (see [Example "Conversion from Cartesian to polar representation of complex data"](#) and [Example "Conversion from polar to Cartesian representation of complex data"](#)).

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

FFT Functions

The fast Fourier transform function library of Intel MKL provides one-dimensional, two-dimensional, and multi-dimensional transforms (of up to seven dimensions) and offers both Fortran and C interfaces for all transform functions.

[Table "FFT Functions in Intel MKL"](#) lists FFT functions implemented in Intel MKL:

FFT Functions in Intel MKL

Function Name	Operation
Descriptor Manipulation Functions	
DftiCreateDescriptor	Allocates the descriptor data structure and initializes it with default configuration values.
DftiCommitDescriptor	Performs all initialization for the actual FFT computation.
DftiFreeDescriptor	Frees memory allocated for a descriptor.
DftiCopyDescriptor	Makes a copy of an existing descriptor.
FFT Computation Functions	
DftiComputeForward	Computes the forward FFT.
DftiComputeBackward	Computes the backward FFT.
Descriptor Configuration Functions	
DftiSetValue	Sets one particular configuration parameter with the specified configuration value.

Function Name	Operation
<code>DftiGetValue</code>	Gets the value of one particular configuration parameter.
Status Checking Functions	
<code>DftiErrorClass</code>	Checks if the status reflects an error of a predefined class.
<code>DftiErrorMessage</code>	Translates the numeric value of an error status into a message.

FFT Interface

The Intel MKL FFT functions are provided with the Fortran and C interfaces. Fortran 95 is required because it offers features that have no counterpart in FORTRAN 77.

NOTE

The Fortran interface of the FFT computation functions requires one-dimensional data arrays for any dimension of FFT problem. For multidimensional transforms, pass the address of the first column of the multidimensional data to the computation functions.

The materials presented in this chapter assume the availability of native complex types in C as they are specified in C9X.

To use the FFT functions, you need to access the module `MKL_DFTI` through the Fortran `use` statement or include `mkl_dfti.h` in your C code.

The Fortran interface provides a derived type `DFTI_DESCRIPTOR`, named constants representing various names of configuration parameters and their possible values, and overloaded functions through the generic functionality of Fortran 95.

The C interface provides the `DFTI_DESCRIPTOR_HANDLE` type, named constants of two enumeration types `DFTI_CONFIG_PARAM` and `DFTI_CONFIG_VALUE`, and functions, some of which accept different numbers of input arguments.

NOTE

The current version of the library may not support some of the FFT functions or functionality described in the subsequent sections of this chapter. You can find the complete list of the implementation-specific exceptions in the Intel MKL Release Notes.

For the main categories of Intel MKL FFT functions, see [FFT Functions](#).

Computing an FFT

You can find code examples that compute transforms in the [Fourier Transform Functions Code Examples](#) section in the Code Examples appendix.

Usually you can compute an FFT by five function calls (refer to the [usage model](#) for details). A single data structure, the descriptor, stores configuration parameters that can be changed independently.

The descriptor data structure, when created, contains information about the length and domain of the FFT to be computed, as well as the setting of several configuration parameters. Default settings for some of these parameters are as follows:

- Scale factor: none
- Number of data sets: one
- Data storage: contiguous
- Placement of results: in-place (the computed result overwrites the input data)

The default settings can be changed one at a time through the function `DftiSetValue` as illustrated in [Example "Changing Default Settings \(Fortran\)"](#) and [Example "Changing Default Settings \(C\)"](#).

Configuration Settings

Each of the configuration parameters is identified by a named constant in the `MKL_DFTI` module. In C, these named constants have the enumeration type `DFTI_CONFIG_PARAM`.

All the Intel MKL FFT configuration parameters are readable. Some of them are read-only, while others can be set using the `DftiCreateDescriptor` or `DftiSetValue` function.

Values of the configuration parameters fall into the following groups:

- Values that have native data types. For example, the number of simultaneous transforms requested has an integer value, while the scale factor for a forward transform is a single-precision number.
- Values that are discrete in nature and are provided in the `MKL_DFTI` module as named constants. For example, the domain of the forward transform requires values to be named constants. In C, the named constants for configuration values have the enumeration type `DFTI_CONFIG_VALUE`.

[Table "Configuration Parameters"](#) summarises the information on configuration parameters, along with their types and values. For more details of each configuration parameter, see the subsection describing this parameter.

Configuration Parameters

Configuration Parameter	Type/Value	Comments
<i>Most common configuration parameters, no default, must be set explicitly by <code>DftiCreateDescriptor</code></i>		
<code>DFTI_PRECISION</code>	Named constant <code>DFTI_SINGLE</code> or <code>DFTI_DOUBLE</code>	Precision of the computation.
<code>DFTI_FORWARD_DOMAIN</code>	Named constant <code>DFTI_COMPLEX</code> or <code>DFTI_REAL</code>	Type of the transform.
<code>DFTI_DIMENSION</code>	Integer scalar	Dimension of the transform.
<code>DFTI_LENGTH</code>	Integer scalar/array	Lengths of each dimension.
<i>Common configuration parameters, settable by <code>DftiSetValue</code></i>		
<code>DFTI_PLACEMENT</code>	Named constant <code>DFTI_INPLACE</code> or <code>DFTI_NOT_INPLACE</code>	Defines whether the result overwrites the input data. Default value: <code>DFTI_INPLACE</code> .
<code>DFTI_FORWARD_SCALE</code>	Floating-point scalar	Scale factor for the forward transform. Default value: 1.0. Precision of the value should be the same as defined by <code>DFTI_PRECISION</code> .
<code>DFTI_BACKWARD_SCALE</code>	Floating-point scalar	Scale factor for the backward transform. Default value: 1.0. Precision of the value should be the same as defined by <code>DFTI_PRECISION</code> .
<code>DFTI_NUMBER_OF_USER_THREADS</code>	Integer scalar	This configuration parameter is no longer used and kept for compatibility with previous versions of Intel MKL.

Configuration Parameter	Type/Value	Comments
DFTI_THREAD_LIMIT	Integer scalar	Limits the number of threads for the DftiComputeForward and DftiComputeBackward functions to use. Default value: 0. In case of the default value, call mkl_domain_set_num_threads to limit the number of threads for the FFT computation functions.
DFTI_DESCRIPTOR_NAME	Character string	Assigns a name to a descriptor. Assumed length of the string is DFTI_MAX_NAME_LENGTH. Default value: empty string.
<i>Data layout configuration parameters for single and multiple transforms. Settable by DftiSetValue</i>		
DFTI_INPUT_STRIDES	Integer array	Defines the input data layout.
DFTI_OUTPUT_STRIDES	Integer array	Defines the output data layout.
DFTI_NUMBER_OF_TRANSFORMS	Integer scalar	Number of transforms. Default value: 1.
DFTI_INPUT_DISTANCE	Integer scalar	Defines the distance between input data sets for multiple transforms. Default value: 0.
DFTI_OUTPUT_DISTANCE	Integer scalar	Defines the distance between output data sets for multiple transforms. Default value: 0.
DFTI_COMPLEX_STORAGE	Named constant DFTI_COMPLEX_COMPLEX or DFTI_REAL_REAL	Defines whether the real and imaginary parts of data for a complex transform are interleaved in one array or split in two arrays. Default value: DFTI_COMPLEX_COMPLEX.
DFTI_REAL_STORAGE	Named constant DFTI_REAL_REAL	Defines how real data for a real transform is stored. Only the DFTI_REAL_REAL value is supported.
DFTI_CONJUGATE_EVEN_STORAGE	Named constant DFTI_COMPLEX_COMPLEX or DFTI_COMPLEX_REAL	Defines whether the complex data in the backward domain of a real transform is stored as complex elements or as real elements. For the default value, see the detailed description.
DFTI_PACKED_FORMAT	Named constant DFTI_CCE_FORMAT, DFTI_CCS_FORMAT, DFTI_PACK_FORMAT, or DFTI_PERM_FORMAT	Defines the layout of real elements in the backward domain of a one-dimensional or two-dimensional real transform.

Advanced configuration parameters, settable by [DftiSetValue](#)

Configuration Parameter	Type/Value	Comments
DFTI_WORKSPACE	Named constant DFTI_ALLOW or DFTI_AVOID	Defines whether the library should prefer algorithms using additional memory. Default value: DFTI_ALLOW.
DFTI_ORDERING	Named constant DFTI_ORDERED or DFTI_BACKWARD_SCRAMBLE	Defines whether the result of a complex transform is ordered or permuted. Default value: DFTI_ORDERED.
<i>Read-Only configuration parameters</i>		
DFTI_COMMIT_STATUS	Named constant DFTI_UNCOMMITTED or DFTI_COMMITED	Readiness of the descriptor for computation.
DFTI_VERSION	String	Version of Intel MKL. Assumed length of the string is DFTI_VERSION_LENGTH.

See Also

[Configuring and Computing an FFT in Fortran](#)

[Configuring and Computing an FFT in C/C++](#)

DFTI_PRECISION

The configuration parameter `DFTI_PRECISION` denotes the floating-point precision in which the transform is to be carried out. A setting of `DFTI_SINGLE` stands for single precision, and a setting of `DFTI_DOUBLE` stands for double precision. The data must be presented in this precision, the computation is carried out in this precision, and the result is delivered in this precision.

`DFTI_PRECISION` does not have a default value. Set it explicitly by calling the `DftiCreateDescriptor` function.

NOTE

Fortran module `MKL_DFTI` also defines named constants `DFTI_SINGLE_R` and `DFTI_DOUBLE_R`, with the same semantics as `DFTI_SINGLE` and `DFTI_DOUBLE`, respectively. Do not use these constants to set the `DFTI_PRECISION` configuration parameter. Use them only as described in section [DftiCreateDescriptor](#).

To better understand configuration of the precision of transforms, you can refer to these examples in your Intel MKL directory:

C:

```
./examples/dftc/source/basic_sp_complex_dft_1d.c
./examples/dftc/source/basic_dp_complex_dft_1d.c
```

Fortran:

```
./examples/dftf/source/basic_sp_complex_dft_1d.f90
./examples/dftf/source/basic_dp_complex_dft_1d.f90
```

See Also

[DFTI_FORWARD_DOMAIN](#)

[DFTI_DIMENSION, DFTI_LENGTHS](#)

[DftiCreateDescriptor](#)

DFTI_FORWARD_DOMAIN

The general form of a discrete Fourier transform is

$$Z_{k_1, k_2, \dots, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \cdots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1, j_2, \dots, j_d} \exp \left(\delta i 2\pi \sum_{l=1}^d j_l k_l / n_l \right)$$

for $k_l = 0, \dots, n_l-1$ ($l = 1, \dots, d$), where σ is a scale factor, $\delta = -1$ for the forward transform, and $\delta = +1$ for the backward transform.

The Intel MKL implementation of the FFT algorithm, used for fast computation of discrete Fourier transforms, supports forward transforms on input sequences of two domains, as specified by the `DFTI_FORWARD_DOMAIN` configuration parameter: general complex-valued sequences (`DFTI_COMPLEX` domain) and general real-valued sequences (`DFTI_REAL` domain). The forward transform maps the forward domain to the corresponding backward domain, as shown in [Table "Correspondence of Forward and Backward Domain"](#).

The conjugate-even domain covers complex-valued sequences with the symmetry property:

$$x(k_1, k_2, \dots, k_d) = \text{conjugate}(x(n_1 - k_1, n_2 - k_2, \dots, n_d - k_d)),$$

where the index arithmetic is performed modulo respective size, that is,

$$x(\dots, expr_s, \dots) \equiv x(\dots, \text{mod}(expr_s, n_s), \dots),$$

and therefore

$$x(\dots, n_s, \dots) \equiv x(\dots, 0, \dots).$$

Due to this property of conjugate-even sequences, only a part of such sequence is stored in the computer memory, as described in [DFTI_CONJUGATE_EVEN_STORAGE](#).

[Correspondence of Forward and Backward Domain](#)

Forward Domain	Implied Backward Domain
Complex (<code>DFTI_COMPLEX</code>)	Complex (<code>DFTI_COMPLEX</code>)
Real (<code>DFTI_REAL</code>)	Conjugate-even

`DFTI_FORWARD_DOMAIN` does not have a default value. Set it explicitly by calling the `DftiCreateDescriptor` function.

To better understand usage of the `DFTI_FORWARD_DOMAIN` configuration parameter, you can refer to these examples in your Intel MKL directory:

C:

```
./examples/dftc/source/basic_sp_complex_dft_1d.c
./examples/dftc/source/basic_sp_real_dft_1d.c
```

Fortran:

```
./examples/dftf/source/basic_sp_complex_dft_1d.f90
./examples/dftf/source/basic_sp_real_dft_1d.f90
```

[See Also](#)

[DFTI_PRECISION](#)

[DFTI_DIMENSION, DFTI_LENGTHS](#)

[DftiCreateDescriptor](#)

DFTI_DIMENSION, DFTI_LENGTHS

The dimension of the transform is a positive integer value represented in an integer scalar of `Integer` data type in Fortran and `MKL_LONG` data type in C. For a one-dimensional transform, the transform length is specified by a positive integer value represented in an integer scalar of `Integer` data type in Fortran and `MKL_LONG` data type in C. For multi-dimensional (≥ 2) transform, the lengths of each of the dimensions are supplied in an integer array (of `Integer` data type in Fortran and `MKL_LONG` data type in C).

`DFTI_DIMENSION` and `DFTI_LENGTHS` do not have a default value. To set them, use the `DftiCreateDescriptor` function and not the `DftiSetValue` function.

To better understand usage of the `DFTI_DIMENSION` and `DFTI_LENGTHS` configuration parameters, you can refer to basic examples of one-, two-, and three-dimensional transforms in your Intel MKL directory. Naming conventions for the examples are self-explanatory. For example, refer to these examples of single-precision two-dimensional transforms:

C:

```
./examples/dftc/source/basic_sp_real_dft_2d.c
./examples/dftc/source/basic_sp_complex_dft_2d.c
```

Fortran:

```
./examples/dftf/source/basic_sp_real_dft_2d.f90
./examples/dftf/source/basic_sp_complex_dft_2d.f90
```

See Also

[DFTI_FORWARD_DOMAIN](#)[DFTI_PRECISION](#)[DftiCreateDescriptor](#)[DftiSetValue](#)

DFTI_PLACEMENT

By default, the computational functions overwrite the input data with the output result. That is, the default setting of the configuration parameter `DFTI_PLACEMENT` is `DFTI_INPLACE`. You can change that by setting it to `DFTI_NOT_INPLACE`.

NOTE

The data sets have no common elements.

To better understand usage of the `DFTI_PLACEMENT` configuration parameter, refer to the following examples in your Intel MKL directory:

C:

```
./examples/dftc/source/config_placement.c
```

Fortran:

```
./examples/dftf/source/config_placement.f90
```

See Also

[DftiSetValue](#)

DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE

The forward transform and backward transform are each associated with a scale factor σ of its own having the default value of 1. You can specify the scale factors using one or both of the configuration parameters `DFTI_FORWARD_SCALE` and `DFTI_BACKWARD_SCALE`. For example, for a one-dimensional transform of length n , you can use the default scale of 1 for the forward transform and set the scale factor for the backward transform to be $1/n$, thus making the backward transform the inverse of the forward transform.

Set the scale factor configuration parameter using a real floating-point data type of the same precision as the value for `DFTI_PRECISION`.

NOTE

For inquiry of the scale factor with the `DftiGetValue` function in C, the `config_val` parameter must have the same floating-point precision as the descriptor.

See Also

`DftiSetValue`
`DFTI_PRECISION`
`DftiGetValue`

DFTI_NUMBER_OF_USER_THREADS

The `DFTI_NUMBER_OF_USER_THREADS` configuration parameter is no longer used and kept for compatibility with previous versions of Intel MKL.

See Also

`DftiSetValue`

DFTI_THREAD_LIMIT

In some situations you may need to limit the number of threads that the `DftiComputeForward` and `DftiComputeBackward` functions use. For example, if more than one thread calls Intel MKL, it might be important that the thread calling these functions does not oversubscribe computing resources (CPU cores). Similarly, a known limit of the maximum number of threads to be used in computations might help the `DftiCommitDescriptor` function to select a more optimal computation method.

Set the parameter `DFTI_THREAD_LIMIT` as follows:

- To a positive number, to specify the maximum number of threads to be used by the compute functions.
- To zero (the default value), to use the maximum number of threads specified by threading control functions `mkl_set_num_threads_local`, `mkl_set_num_threads`, `mkl_domain_set_num_threads`, and/or `omp_set_num_threads`.

On an attempt to set a negative value, the `DftiSetValue` function returns an error and does not update the descriptor.

The value of the `DFTI_THREAD_LIMIT` configuration parameter returned by the `DftiGetValue` function is defined as follows:

- 1 if Intel MKL runs in the sequential mode
 - Depends of the commit status of the descriptor if Intel MKL runs in a threaded mode:
-

Commit Status	Value
Not committed	The value of <code>DFTI_THREAD_LIMIT</code> set in a previous call to the <code>DftiSetValue</code> function or the default value
Committed	The minimum of the value returned by the <code>mkl_domain_get_max_threads(MKL_DOMAIN_FFT)</code> function call and the optimal number of threads computed by the <code>DftiCommitDescriptor</code> function

To better understand usage of the `DFTI_THREAD_LIMIT` configuration parameter, refer to the following examples in your Intel MKL directory:

C:

```
./examples/dftc/source/config_thread_limit.c
```

Fortran:

```
./examples/dftf/source/config_thread_limit.f90
```

See Also

[DftiGetValue](#)
[DftiSetValue](#)
[DftiCommitDescriptor](#)
[DftiComputeForward](#)
[DftiComputeBackward](#)
[Threading Control](#)
[DFTI_COMMIT_STATUS](#)

DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES

The FFT interface provides configuration parameters that define the layout of multidimensional data in the computer memory. For d -dimensional data set X defined by dimensions $N_1 \times N_2 \times \dots \times N_d$, the layout describes where a particular element $X(k_1, k_2, \dots, k_d)$ of the data set is located. The memory address of the element $X(k_1, k_2, \dots, k_d)$ is expressed by the formula

$$\begin{aligned} \text{address of } X(k_1, k_2, \dots, k_d) &= \text{address of } X(0, 0, \dots, 0) + \text{offset} \\ &= \text{address of } X(0, 0, \dots, 0) + s_0 + k_1*s_1 + k_2*s_2 + \dots + k_d*s_d, \end{aligned}$$

where s_0 is the displacement and s_1, \dots, s_d are generalized strides. The configuration parameters [DFTI_INPUT_STRIDES](#) and [DFTI_OUTPUT_STRIDES](#) enable you to get and set these values. The configuration value is an array of values (s_0, s_1, \dots, s_d) of [INTEGER](#) data type in Fortran and [MKL_LONG](#) data type in C.

The offset is counted in elements of the data type defined by the descriptor configuration (rather than by the type of the variable passed to the computation functions). Specifically, the

[DFTI_FORWARD_DOMAIN](#), [DFTI_COMPLEX_STORAGE](#), and [DFTI_CONJUGATE_EVEN_STORAGE](#) configuration parameters define the type of the elements as shown in [Table "Assumed Element Types of the Input/Output Data"](#):

Assumed Element Types of the Input/Output Data

Descriptor Configuration	Element Type in the Forward Domain	Element Type in the Backward Domain
DFTI_FORWARD_DOMAIN=DFTI_COMPLEX DFTI_COMPLEX_STORAGE=DFTI_COMPLEX_COMPLEX	Complex	Complex
DFTI_FORWARD_DOMAIN=DFTI_COMPLEX DFTI_COMPLEX_STORAGE=DFTI_REAL_REAL	Real	Real
DFTI_FORWARD_DOMAIN=DFTI_REAL DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL	Real	Real
DFTI_FORWARD_DOMAIN=DFTI_REAL DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX	Real	Complex

The [DFTI_INPUT_STRIDES](#) configuration parameter defines the layout of the input data, while the element type is defined by the forward domain for the [DftiComputeForward](#) function and by the backward domain for the [DftiComputeBackward](#) function. The [DFTI_OUTPUT_STRIDES](#) configuration parameter defines the layout of the output data, while the element type is defined by the backward domain for the [DftiComputeForward](#) function and by the forward domain for [DftiComputeBackward](#) function.

NOTE

The [DFTI_INPUT_STRIDES](#) and [DFTI_OUTPUT_STRIDES](#) configuration parameters define the layout of input and output data, and not the forward-domain and backward-domain data. If the data layouts in forward domain and backward domain differ, set [DFTI_INPUT_STRIDES](#) and [DFTI_OUTPUT_STRIDES](#) explicitly and then commit the descriptor before calling computation functions.

For in-place transforms (`DFTI_PLACEMENT=DFTI_INPLACE`), the configuration set by `DFTI_OUTPUT_STRIDES` is ignored when the element types in the forward and backward domains are the same. If they are different, set `DFTI_OUTPUT_STRIDES` explicitly (even though the transform is in-place). Ensure a consistent configuration for in-place transforms, that is, the locations of the first elements on input and output must coincide in each dimension.

The FFT interface supports both positive and negative stride values. If you use negative strides, set the displacement of the data as follows:

$$s_0 = \sum_{i=1}^d (N_i - 1) \cdot \max(-s_i, 0).$$

The default setting of strides in a general multi-dimensional case assumes that the array that contains the data has no padding. The order of the strides depends on the programming language. For example:

```
/* C/C++ */
MKL_LONG dims[] = { nd, ..., n2, n1 };
DftiCreateDescriptor( &hand, precision, domain, d, dims );
// The above call assumes data declaration: type X[nd]...[n2][n1]
// Default strides are { 0, nd*...*n2*n1, ..., n2*n1, n1, 1 }
```

```
! Fortran
INTEGER :: dims(d) = [n1, n2, ..., nd]
status = DftiCreateDescriptor( hand, precision, domain, d, dims )
! The above call assumes data declaration: type X(n1,n2,...,nd)
! Default strides are [ 0, 1, n1, n1*n2, ..., n1*n2*...*nd]
```

Note that in case of a real FFT (`DFTI_FORWARD_DOMAIN=DFTI_REAL`), where different data layouts in the backward domain are available (see [DFTI_PACKED_FORMAT](#)), the default value of the strides is not intuitive for the recommended CCE format (configuration setting

`DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX`). In case of an *in-place* real transform with the CCE format, set the strides explicitly, as follows:

```
/* C/C++ */
MKL_LONG dims[] = { nd, ..., n2, n1 };
MKL_LONG rstrides[] = { 0, 2*nd*...*n2*(n1/2+1), ..., 2*n2*(n1/2+1), 2*(n1/2+1), 1 };
MKL_LONG cstrides[] = { 0, nd*...*n2*(n1/2+1), ..., n2*(n1/2+1), (n1/2+1), 1 };
DftiCreateDescriptor( &hand, precision, DFTI_REAL, d, dims );
DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX);
// Set the strides appropriately for forward/backward transform
```

```
! Fortran
INTEGER :: dims(d) = [n1, n2, ..., nd]
INTEGER :: rstrides(1+d) = [0, 1, 2*(n1/2+1), 2*(n1/2+1)*n2, ... ]
INTEGER :: cstrides(1+d) = [0, 1, (n1/2+1), (n1/2+1)*n2, ... ]
status = DftiCreateDescriptor( hand, precision, domain, d, dims )
status = DftiSetValue( hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX)
! Set the strides appropriately for forward/backward transform
```

To better understand configuration of strides, you can also refer to these examples in your Intel MKL directory:

C:

```
./examples/dftc/source/basic_sp_real_dft_2d.c
./examples/dftc/source/basic_sp_real_dft_3d.c
./examples/dftc/source/basic_dp_real_dft_2d.c
./examples/dftc/source/basic_dp_real_dft_3d.c
```

Fortran:

```
./examples/dftf/source/basic_sp_real_dft_2d.f90
./examples/dftf/source/basic_sp_real_dft_3d.f90
./examples/dftf/source/basic_dp_real_dft_2d.f90
./examples/dftf/source/basic_dp_real_dft_3d.f90
```

See Also

[DFTI_FORWARD_DOMAIN](#)
[DFTI_PLACEMENT](#)
[DftiSetValue](#)
[DftiCommitDescriptor](#)
[DftiComputeForward](#)
[DftiComputeBackward](#)

DFTI_NUMBER_OF_TRANSFORMS

In some situations, you may need to perform a number of identical FFTs. You can do this with a single call to a `DftiCompute*` function if the value of the `DFTI_NUMBER_OF_TRANSFORMS` configuration parameter is set to the actual number of the transforms. The default value of this parameter is one. You can set this parameter to a positive integer value using the `Integer` data type in Fortran and `MKL_LONG` data type in C. When setting the number of transforms to a value greater than one, you also need to specify the distance between the input and output data sets using one of the `DFTI_INPUT_DISTANCE` and `DFTI_OUTPUT_DISTANCE` configuration parameters or both.

-
- The data sets to be transformed must not have common elements.
 - All the sets of data must be located within the same memory block.
-

To better understand usage of the `DFTI_NUMBER_OF_TRANSFORMS` configuration parameter, refer to the following examples in your Intel MKL directory:

C:

```
./examples/dftc/source/config_number_of_transforms.c
```

Fortran:

```
./examples/dftf/source/config_number_of_transforms.f90
```

See Also

[FFT Computation Functions](#)
[DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE](#)
[DftiSetValue](#)

DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE

The FFT interface in Intel MKL enables computation of multiple transforms. To compute multiple transforms, you need to specify the data distribution of the multiple sets of data. The distance between the first data elements of consecutive data sets, `DFTI_INPUT_DISTANCE` for input data or `DFTI_OUTPUT_DISTANCE` for output data, specifies the distribution. The configuration setting is a value of `INTEGER` data type in Fortran and `MKL_LONG` data type in C.

The default value for both configuration settings is one. You must set this parameter explicitly if the number of transforms is greater than one (see [DFTI_NUMBER_OF_TRANSFORMS](#)).

The distance is counted in elements of the data type defined by the descriptor configuration (rather than by the type of the variable passed to the computation functions). Specifically, the `DFTI_FORWARD_DOMAIN`, `DFTI_COMPLEX_STORAGE`, and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters define the type of the elements as shown in [Table "Assumed Element Types of the Input/Output Data"](#).

NOTE

The configuration parameters `DFTI_INPUT_DISTANCE` and `DFTI_OUTPUT_DISTANCE` define the distance within input and output data, and not within the forward-domain and backward-domain data. If the distances in the forward and backward domains differ, set `DFTI_INPUT_DISTANCE` and `DFTI_OUTPUT_DISTANCE` explicitly and then commit the descriptor before calling computation functions.

For in-place transforms (`DFTI_PLACEMENT=DFTI_INPLACE`), the configuration set by `DFTI_OUTPUT_DISTANCE` is ignored when the element types in the forward and backward domains are the same. If they are different, set `DFTI_OUTPUT_DISTANCE` explicitly (even though the transform is in-place). Ensure a consistent configuration for in-place transforms, that is, the locations of the data sets on input and output must coincide.

The following example in C and Fortran illustrates setting of the `DFTI_INPUT_DISTANCE` configuration parameter:

```
/* C/C++ */
MKL_LONG dims[] = { nd, ..., n2, n1 };
MKL_LONG distance = nd*...*n2*n1;
DftiCreateDescriptor( &hand, precision, DFTI_COMPLEX, d, dims );
DftiSetValue( hand, DFTI_NUMBER_OF_TRANSFORMS, (MLK_LONG)howmany );
DftiSetValue( hand, DFTI_INPUT_DISTANCE, distance );

! Fortran
INTEGER :: dims(d) = [n1, n2, ..., nd]
INTEGER :: distance = n1*n2*...*nd
status = DftiCreateDescriptor( hand, precision, DFTI_COMPLEX, d, dims)
status = DftiSetValue( hand, DFTI_NUMBER_OF_TRANSFORMS, howmany )
status = DftiSetValue( hand, DFTI_INPUT_DISTANCE, distance );
```

To better understand configuration of the distances, you can also refer to the following examples in your Intel MKL directory:

C:

```
./examples/dftc/source/config_number_of_transforms.c
```

Fortran:

```
./examples/dftf/source/config_number_of_transforms.f90
```

See Also

[DFTI_PLACEMENT](#)

[DftiSetValue](#)

[DftiCommitDescriptor](#)

[DftiComputeForward](#)

[DftiComputeBackward](#)

DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE

Depending on the value of the `DFTI_FORWARD_DOMAIN` configuration parameter, the implementation of FFT supports several storage schemes for input and output data (see document [3] for the rationale behind the definition of the storage schemes). The data elements are placed within contiguous memory blocks, defined with generalized strides (see `DFTI_INPUT_STRIDES`, `DFTI_OUTPUT_STRIDES`). For multiple transforms, all sets of data should be located within the same memory block, and the data sets should be placed at the same distance from each other (see `DFTI_NUMBER_OF_TRANSFORMS` and `DFTI_INPUT_DISTANCE`, `DFTI_OUTPUT_DISTANCE`).

TIP

In C/C++, avoid setting up multidimensional arrays with lists of pointers to one-dimensional arrays. Instead use a one-dimensional array with the explicit indexing to access the data elements.

[FFT Examples](#) demonstrate the usage of storage formats in both C and Fortran.

DFTI_COMPLEX_STORAGE: storage schemes for a complex domain

For the DFTI_COMPLEX forward domain, both input and output sequences belong to a complex domain. In this case, the configuration parameter DFTI_COMPLEX_STORAGE can have one of the two values: DFTI_COMPLEX_COMPLEX (default) or DFTI_REAL_REAL.

NOTE

In the Intel MKL FFT interface, storage schemes for a forward complex domain and the respective backward complex domain are the same.

With DFTI_COMPLEX_COMPLEX storage, complex-valued data sequences are referenced by a single complex parameter (array) AZ so that a complex-valued element z_{k_1, k_2, \dots, k_d} of the m-th d-dimensional sequence is located at $AZ[m * \text{distance} + \text{stride}_0 + k_1 * \text{stride}_1 + k_2 * \text{stride}_2 + \dots + k_d * \text{stride}_d]$ as a structure consisting of the real and imaginary parts.

The following code illustrates usage of the DFTI_COMPLEX_COMPLEX storage:

```
! Fortran
complex :: AZ(N1,N2,N3,M) ! sizes and number of transforms
...
! on input: Z{k1,k2,k3,m} = AZ(k1,k2,k3,m)
status = DftiComputeForward( desc, AZ(:,1,1,1) )
! on output: Z{k1,k2,k3,m} = AZ(k1,k2,k3,m)

// C/C++
complex *AZ = malloc( N1*N2*N3*M * sizeof(AZ[0]) );
MKL_LONG ios[4], iodist; // input/output strides and distance
...
// on input: Z{k1,k2,k3,m}
// = AZ[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
status = DftiComputeForward( desc, AZ );
// on output: Z{k1,k2,k3,m}
// = AZ[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
```

With the DFTI_REAL_REAL storage, complex-valued data sequences are referenced by two real parameters AR and AI so that a complex-valued element z_{k_1, k_2, \dots, k_d} of the m-th sequence is computed as $AR[m * \text{distance} + \text{stride}_0 + k_1 * \text{stride}_1 + k_2 * \text{stride}_2 + \dots + k_d * \text{stride}_d] + \sqrt{-1} * AI[m * \text{distance} + \text{stride}_0 + k_1 * \text{stride}_1 + k_2 * \text{stride}_2 + \dots + k_d * \text{stride}_d]$.

The following code illustrates usage of the DFTI_REAL_REAL storage:

```
! Fortran
real :: AR(N1,N2,N3,M), AI(N1,N2,N3,M)
...
! on input: Z{k1,k2,k3,m} = cmplx(AR(k1,k2,k3,m),AI(k1,k2,k3,m))
status = DftiComputeForward( desc, AR(:,1,1,1), AI(:,1,1,1) )
! on output: Z{k1,k2,k3,m} = cmplx(AR(k1,k2,k3,m),AI(k1,k2,k3,m))

// C/C+
float *AR = malloc( N1*N2*N3*M * sizeof(AR[0]) );
float *AI = malloc( N1*N2*N3*M * sizeof(AI[0]) );
MKL_LONG ios[4], iodist; // input/output strides and distance
...
// on input: Z{k1,...,kd,m}
// = AR[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
// + I*AI[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
status = DftiComputeForward( desc, AR, AI );
```

```
// on output: Z{k1,...,kd,m}
// = AR[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
// + I*AI[ ios[0] + k1*ios[1] + k2*ios[2] + k3*ios[3] + m*iodist ]
```

DFTI_REAL_STORAGE: storage schemes for a real domain

The Intel MKL FFT interface supports only one configuration value for this storage scheme: DFTI_REAL_REAL. With the DFTI_REAL_REAL storage, real-valued data sequences in a real domain are referenced by one real parameter AR so that real-valued element of the m -th sequence is located as $AR[m*distance + stride_0 + k_1*stride_1 + k_2*stride_2 + \dots + k_d*strided]$.

DFTI_CONJUGATE_EVEN_STORAGE: storage scheme for a conjugate-even domain

The Intel MKL FFT interface supports two configuration values for this parameter: DFTI_COMPLEX_REAL (default) and DFTI_COMPLEX_COMPLEX. In both cases, the conjugate-even symmetry of the data enables storing only about a half of the whole mathematical result, so that one part of it can be directly referenced in the memory while the other part can be reconstructed depending on the selected storage configuration.

With the DFTI_COMPLEX_REAL storage, the complex-valued data sequences in the conjugate-even domain can be reconstructed as described in section [DFTI_PACKED_FORMAT](#).

Important

Although DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL is the default setting for the DFTI_REAL forward domain, avoid using this storage scheme because it is supported only for one- and two-dimensional transforms and will be deprecated in future.

With the DFTI_COMPLEX_COMPLEX storage, the complex-valued data sequences in the conjugate-even domain are referenced by one complex parameter AZ so that a complex-valued element Z_{k_1, k_2, \dots, k_d} of the m -th sequence can be referenced or reconstructed as described below.

Important

Use the DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX configuration setting, which will become the default in future. This setting is supported for all transform ranks, provides a uniform pattern for reconstructing the entire conjugate-even sequence from the part of it that is actually stored, and is compatible with data layouts supported by other FFT libraries, such as FFTW. This storage scheme disregards the setting of DFTI_PACKED_FORMAT.

Consider a d-dimensional real-to-complex transform

$$Z_{k_1, k_2, \dots, k_d} \equiv \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_d=0}^{N_d-1} R_{n_1, n_2, \dots, n_d} e^{\frac{-2\pi i}{N_1} k_1 \cdot n_1} \cdots e^{\frac{-2\pi i}{N_d} k_d \cdot n_d}$$

Because the input sequence R is real-valued, the mathematical result Z has conjugate-even symmetry:

$$Z_{k_1, k_2, \dots, k_d} = \text{conjugate}(Z_{N_1-k_1, N_2-k_2, \dots, N_d-k_d}),$$

where index arithmetic is performed modulo the length of the respective dimension. Obviously, the first element of the result is real-valued:

$$Z_{0, 0, \dots, 0} = \text{conjugate}(Z_{0, 0, \dots, 0}).$$

For dimensions with even lengths, the other elements are real-valued too. For example, if N_s is even,

$$Z_{0, 0, \dots, N_s/2, 0, \dots, 0} = \text{conjugate}(Z_{0, 0, \dots, N_s/2, 0, \dots, 0}).$$

With the conjugate-even symmetry, approximately a half of the result suffices to fully reconstruct it. For an arbitrary dimension h , it suffices to store elements $z_{k_1, \dots, k_h, \dots, k_d}$ for the following indices:

- $k_h = 0, \dots, [N_h / 2]$
- $k_i = 0, \dots, N_i - 1$, where $i = 1, \dots, d$ and $i \neq h$

The symmetry property enables reconstructing the remaining elements: for $k_h = [N_h / 2] + 1, \dots, N_h - 1$. In the Intel MKL FFT interface, the halved dimension is the first dimension in Fortran and the last dimension in C/C++.

The following code illustrates usage of the `DFTI_COMPLEX_COMPLEX` storage for a conjugate-even domain:

```

! Fortran
real :: AR(N1,N2,M)           ! Array containing values of R
complex :: AZ(N1/2+1,N2,M)    ! Array containing values of Z
...
! on input: R{k1,k2,m} = AR(k1,k2,m)
status = DftiComputeForward( desc, AR(:,1,1), AZ(:,1,1) )
! on output:
! for k1=1 ... N1/2+1: Z{k1,k2,m} = AZ(k1,k2,m)
! for k1=N1/2+2 ... N1: Z{k1,k2,m} = conj(AZ(mod(N1-k1+1,N1)+1,mod(N2-k2+1,N2)+1,m))
// C/C++
float *AR = malloc( N1*N2*M * sizeof(AR[0]) );
complex *AZ = malloc( N1*(N2/2+1)*M * sizeof(AZ[0]) );
MKL_LONG is[3], os[3], idist, odist; // input and output strides and distance
...
// on input: R{k1,k2,m}
// = AR[is[0] + k1*is[1] + k2*is[2] + m*idist]
status = DftiComputeForward( desc, R, C );
// on output:
// for k2=0..N2/2:      Z{k1,k2,m} = AZ[os[0]+k1*os[1]+k2*os[2]+m*odist]
// for k2=N2/2+1..N2-1: Z{k1,k2,m} = conj(AZ[os[0]+(N1-k1)%N1*os[1]
//                                         +(N2-k2)%N2*os[2]+m*odist])
// 
```

For the backward transform, the input and output parameters and layouts exchange roles: set the strides describing the layout in the backward/forward domain as input/output strides, respectively. For example:

```

...
status = DftiSetValue( desc, DFTI_INPUT_STRIDES, fwd_domain_strides )
status = DftiSetValue( desc, DFTI_OUTPUT_STRIDES, bwd_domain_strides )
status = DftiCommitDescriptor( desc )
status = DftiComputeForward( desc, ... )
...
status = DftiSetValue( desc, DFTI_INPUT_STRIDES, bwd_domain_strides )
status = DftiSetValue( desc, DFTI_OUTPUT_STRIDES, fwd_domain_strides )
status = DftiCommitDescriptor( desc )
status = DftiComputeBackward( desc, ... )

```

Important

For in-place transforms, ensure the first element of the input data has the same address as the first element of the output data for each dimension.

See Also

[DftiSetValue](#)

[DFTI_PACKED_FORMAT](#)

The result of the forward transform of real data is a conjugate-even sequence. Due to the symmetry property, only a part of the complex-valued sequence is stored in memory. The `DFTI_PACKED_FORMAT` configuration parameter defines how the data is packed. Possible values of `DFTI_PACKED_FORMAT` depend on the values of the `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameter:

- `DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX`.

The only value of `DFTI_PACKED_FORMAT` can be `DFTI_CCE_FORMAT`. You can use this value with transforms of any dimension. For a description of the corresponding packed format, see `DFTI_CONJUGATE_EVEN_STORAGE`.

- `DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL`.

`DFTI_PACKED_FORMAT` can be `DFTI_CCS_FORMAT`, `DFTI_PACK_FORMAT`, or `DFTI_PERM_FORMAT`. You can use these values with one- and two-dimensional transforms only. The corresponding packed formats are described below.

NOTE

Although `DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL` is the default setting for the `DFTI_REAL` forward domain, avoid using this storage scheme because it is supported only for one- and two-dimensional transforms, is incompatible with storage schemes of other FFT libraries, and will be deprecated in future.

DFTI_CCS_FORMAT for One-dimensional Transforms

The following figure illustrates the storage of a one-dimensional (1D) size- N conjugate-even sequence in a real array for the CCS, PACK, and PERM packed formats. The CCS format requires an array of size $N+2$, while the other formats require an array of size N . Zero-based indexing is used.

Storage of a 1D Size- N Conjugate-even Sequence in a Real Array

	$n=$	0	1	2	3	...	$2L-2$	$2L-1$	$2L$	$2L+1$	$2L+2$
CCS, $N=2L$		R_0	0	R_1	I_1	...	R_{L-1}	I_{L-1}	R_L	0	
PACK, $N=2L$		R_0	R_1	I_1	R_2	...	I_{L-1}	R_L			
PERM, $N=2L$		R_0	R_L	R_1	I_1	...	R_{L-1}	I_{L-1}			
CCS, $N=2L+1$		R_0	0	R_1	I_1	...	R_{L-1}	I_{L-1}	R_L	I_L	not used
PACK, $N=2L+1$		R_0	R_1	I_1	R_2	...	I_{L-1}	R_L	I_L		
PERM, $N=2L+1$		R_0	R_1	I_1	R_2	...	I_{L-1}	R_L	I_L		

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in a real-valued array `AC` as illustrated by figure "Storage of a 1D Size- N Conjugate-even Sequence in a Real Array" and can be used to reconstruct the whole conjugate-even sequence as follows:

```

! Fortran
real :: AR(N), AC(N+2)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_CCS_FORMAT )
...
! on input: R{k} = AR(k)
status = DftiComputeForward( desc, AR, AC ) ! real-to-complex FFT
! on output:
! for k=1 ... N/2+1: Z{k} = cmplx( AC(1 + (2*(k-1)+0)),
```

```
!           AC(1 + (2*(k-1)+1)) )
! for k=N/2+2 ... N: Z{k} = cmplx( AC(1 + (2*mod(N-k+1,N)+0)),
!                                -AC(1 + (2*mod(N-k+1,N)+1)) )
```

```
// C/C++
float *AR; // malloc( sizeof(float)*N )
float *AC; // malloc( sizeof(float)*(N+2) )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_CCS_FORMAT );
...
// on input: R{k} = AR[k]
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
// on output:
// for k=0..N/2:      Z{k} =      AC[2*k+0]          + I*AC[2*k+1]
// for k=N/2+1..N-1: Z{k} =      AC[2*(N-k)%N + 0] - I*AC[2*(N-k)%N + 1]
```

DFTI_CCS_FORMAT for Two-dimensional Transforms

The following figure illustrates the storage of a two-dimensional (2D) M -by- N conjugate-even sequence in a real array for the CCS packed format. This format requires an array of size $(M+2)$ -by- $(N+2)$. Row-major layout and zero-based indexing are used. Different colors mark logically separate parts of the result. "n/u" means "not used".

Storage of a 2D M -by- N Conjugate-even Sequence in a Real Array for the CCS Format

	$n=0$	1	2	3	...	$2L-2$	$2L-1$	$2L$	$2L+1$
$m=0$	$R_{0,0}$	0	$R_{0,1}$	$I_{0,1}$...	$R_{0,L-1}$	$I_{0,L-1}$	$R_{0,L}$	0
1	0	0	$R_{1,1}$	$I_{1,1}$...	$R_{1,L-1}$	$I_{1,L-1}$	0	0
2	$R_{1,0}$	0	$R_{2,1}$	$I_{2,1}$...	$R_{2,L-1}$	$I_{2,L-1}$	$R_{1,L}$	0
3	$I_{1,0}$	0	$R_{3,1}$	$I_{3,1}$...	$R_{3,L-1}$	$I_{3,L-1}$	$I_{1,L}$	0
...
$2K-2$	$R_{K-1,0}$	0	$R_{2K-2,1}$	$I_{2K-2,1}$...	$R_{2K-2,L-1}$	$I_{2K-2,L-1}$	$R_{K-1,L}$	0
$2K-1$	$I_{K-1,0}$	0	$R_{2K-1,1}$	$I_{2K-1,1}$...	$R_{2K-1,L-1}$	$I_{2K-1,L-1}$	$I_{K-1,L}$	0
$2K$	$R_{K,0}$	0	n/u	n/u	...	n/u	n/u	$R_{K,L}$	0
$2K+1$	0	0	n/u	n/u	...	n/u	n/u	0	0

CCS, $M=2K$, $N=2L$

	$n=0$	1	2	3	...	$2L-2$	$2L-1$	$2L$	$2L+1$	$2L+2$
$m=0$	$R_{0,0}$	0	$R_{0,1}$	$I_{0,1}$...	$R_{0,L-1}$	$I_{0,L-1}$	$R_{0,L}$	$I_{0,L}$	n/u
1	0	0	$R_{1,1}$	$I_{1,1}$...	$R_{1,L-1}$	$I_{1,L-1}$	$R_{1,L}$	$I_{1,L}$	n/u
2	$R_{1,0}$	0	$R_{2,1}$	$I_{2,1}$...	$R_{2,L-1}$	$I_{2,L-1}$	$R_{2,L}$	$I_{2,L}$	n/u
3	$I_{1,0}$	0	$R_{3,1}$	$I_{3,1}$...	$R_{3,L-1}$	$I_{3,L-1}$	$R_{3,L}$	$I_{3,L}$	n/u
...
$2K-2$	$R_{K-1,0}$	0	$R_{2K-2,1}$	$I_{2K-2,1}$...	$R_{2K-2,L-1}$	$I_{2K-2,L-1}$	$R_{2K-2,L}$	$I_{2K-2,L}$	n/u
$2K-1$	$I_{K-1,0}$	0	$R_{2K-1,1}$	$I_{2K-1,1}$...	$R_{2K-1,L-1}$	$I_{2K-1,L-1}$	$R_{2K-1,L}$	$I_{2K-1,L}$	n/u
$2K$	$R_{K,0}$	0	n/u	n/u	...	n/u	n/u	n/u	n/u	n/u
$2K+1$	0	0	n/u	n/u	...	n/u	n/u	n/u	n/u	n/u

CCS, $M=2K$, $N=2L+1$

	n=0	1	2	3	...	2L-2	2L-1	2L	2L+1
m=0	R _{0,0}	0	R _{0,1}	I _{0,1}	...	R _{0,L-1}	I _{0,L-1}	R _{0,L}	0
1	0	0	R _{1,1}	I _{1,1}	...	R _{1,L-1}	I _{1,L-1}	0	0
2	R _{1,0}	0	R _{2,1}	I _{2,1}	...	R _{2,L-1}	I _{2,L-1}	R _{1,L}	0
3	I _{1,0}	0	R _{3,1}	I _{3,1}	...	R _{3,L-1}	I _{3,L-1}	I _{1,L}	0
...
2K-2	R _{K-1,0}	0	R _{2K-2,1}	I _{2K-2,1}	...	R _{2K-2,L-1}	I _{2K-2,L-1}	R _{K-1,L}	0
2K-1	I _{K-1,0}	0	R _{2K-1,1}	I _{2K-1,1}	...	R _{2K-1,L-1}	I _{2K-1,L-1}	I _{K-1,L}	0
2K	R _{K,0}	0	R _{2K,1}	I _{2K,1}	...	R _{2K,L-1}	I _{2K,L-1}	R _{K,L}	0
2K+1	I _{K,0}	0	n/u	n/u	...	n/u	n/u	I _{K,L}	n/u
2K+2	n/u	n/u	n/u	n/u	...	n/u	n/u	n/u	n/u

CCS, M=2K+1, N=2L

	n=0	1	2	3	...	2L-2	2L-1	2L	2L+1	2L+2
m=0	R _{0,0}	0	R _{0,1}	I _{0,1}	...	R _{0,L-1}	I _{0,L-1}	R _{0,L}	I _{0,L}	n/u
1	0	0	R _{1,1}	I _{1,1}	...	R _{1,L-1}	I _{1,L-1}	R _{1,L}	I _{1,L}	n/u
2	R _{1,0}	0	R _{2,1}	I _{2,1}	...	R _{2,L-1}	I _{2,L-1}	R _{2,L}	I _{2,L}	n/u
3	I _{1,0}	0	R _{3,1}	I _{3,1}	...	R _{3,L-1}	I _{3,L-1}	R _{3,L}	I _{3,L}	n/u
...
2K-2	R _{K-1,0}	0	R _{2K-2,1}	I _{2K-2,1}	...	R _{2K-2,L-1}	I _{2K-2,L-1}	R _{2K-2,L}	I _{2K-2,L}	n/u
2K-1	I _{K-1,0}	0	R _{2K-1,1}	I _{2K-1,1}	...	R _{2K-1,L-1}	I _{2K-1,L-1}	R _{2K-1,L}	I _{2K-1,L}	n/u
2K	R _{K,0}	0	R _{2K,1}	I _{2K,1}	...	R _{2K,L-1}	I _{2K,L-1}	R _{2K,L}	I _{2K,L}	n/u
2K+1	I _{K,0}	0	n/u	n/u	...	n/u	n/u	n/u	n/u	n/u
2K+2	n/u	n/u	n/u	n/u	...	n/u	n/u	n/u	n/u	n/u

CCS, M=2K+1, N=2L+1

The real and imaginary parts of the complex-valued conjugate-even sequence $Z_{k1,k2}$ are located in a real-valued array `AC` as illustrated by figure "[Storage of a 2D M-by-N Conjugate-even Sequence in a Real Array for the CCS Format](#)" and can be used to reconstruct the whole sequence as follows:

```

! Fortran
real :: AR(N1,N2), AC(N1+2,N2+2)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_CCS_FORMAT )
...
! on input: R{k1,k2} = AR(k1,k2)
status = DftiComputeForward( desc, AR(:,1) AC(:,1) ) ! real-to-complex FFT
! on output: Z{k1,k2} = cmplx( re, im ), where
! if (k2 == 1) then
!   if (k1 <= N1/2+1) then
!     re = AC(1+2*(k1-1)+0, 1)
!     im = AC(1+2*(k1-1)+1, 1)
!   else
!     re = AC(1+2*(N1-k1+1)+0, 1)
!     im = -AC(1+2*(N1-k1+1)+1, 1)
!   end if
! else if (k1 == 1) then
!   if (k2 <= N2/2+1) then
!     re = AC(1, 1+2*(k2-1)+0)
!     im = AC(1, 1+2*(k2-1)+1)
!   else
!     re = AC(1, 1+2*(N2-k2+1)+0)
!     im = -AC(1, 1+2*(N2-k2+1)+1)
!   end if
! else if (k1-1 == N1-k1+1) then
!   if (k2 <= N2/2+1) then
!     re = AC(N1+1, 1+2*(k2-1)+0)
!     im = AC(N1+1, 1+2*(k2-1)+1)
!   else
!     re = AC(N1+1, 1+2*(N2-k2+1)+0)
!     im = -AC(N1+1, 1+2*(N2-k2+1)+1)
!   end if
! else if (k1 <= N1/2+1) then
!   re = AC(1+2*(k1-1)+0, k2)
!   im = AC(1+2*(k1-1)+1, k2)
! else
!   re = AC(1+2*(N1-k1+1)+0, 1+N2-k2+1)
!   im = -AC(1+2*(N1-k1+1)+1, 1+N2-k2+1)
! end if

```

```

// C/C++
float *AR; // malloc( sizeof(float)*N1*N2 )
float *AC; // malloc( sizeof(float)*(N1+2)*(N2+2) )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_CCS_FORMAT );
...
// on input: R{k1,k2} = AR[(N2+2)*k1 + k2]
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
// on output: Z{k1,k2} = re + I*im, where
// if (k1==0) {
//   if (k2 <= N2/2) {
//     re = AC[2*k2+0];
//     im = AC[2*k2+1];
//   } else {
//     re = AC[2*(N2-k2)+0];
//   }
// }

```

```

//           im = -AC[2*(N2-k2)+1];
//       }
//   else if (k2==0){
//       if (k1 <= N1/2) {
//           re = AC[(2*k1+0)*(N2+2)];
//           im = AC[(2*k1+1)*(N2+2)];
//       } else {
//           re = AC[(2*(N1-k1)+0)*(N2+2)];
//           im = -AC[(2*(N1-k1)+1)*(N2+2)];
//       }
//   else if (k2 == N2-k2) {
//       if (k1 <= N1/2) {
//           re = AC[(2*k1+0)*(N2+2) + 2*(N2/2)];
//           im = AC[(2*k1+1)*(N2+2) + 2*(N2/2)];
//       } else{
//           re = AC[(2*(N1-k1)+0)*(N2+2) + 2*(N2/2)];
//           im = -AC[(2*(N1-k1)+1)*(N2+2) + 2*(N2/2)];
//       }
//   else if (k2 <= N2/2) {
//       re = AC[k1*(N2+2)+2*k2+0];
//       im = AC[k1*(N2+2)+2*k2+1];
//   } else {
//       re = AC[(N1-k1)*(N2+2)+2*(N2-k2)+0];
//       im = -AC[(N1-k1)*(N2+2)+2*(N2-k2)+1];
//   }

```

DFTI_PACK_FORMAT for One-dimensional Transforms

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in a real-valued array `AC` as illustrated by figure "[Storage of a 1D Size- \$N\$ Conjugate-even Sequence in a Real Array](#)" and can be used to reconstruct the whole conjugate-even sequence as follows:

```

! Fortran
real :: AR(N), AC(N)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PACK_FORMAT )
...
! on input: R{k} = AR(k)
status = DftiComputeForward( desc, AR, AC ) ! real-to-complex FFT
! on output: Z{k} = cmplx( re, im ), where
! if (k == 1) then
!     re = AC(1)
!     im = 0
! else if (k-1 == N-k+1) then
!     re = AC(2*(k-1))
!     im = 0
! else if (k <= N/2+1) then
!     re = AC(2*(k-1)+0)
!     im = AC(2*(k-1)+1)
! else
!     re = AC(2*(N-k+1)+0)
!     im = -AC(2*(N-k+1)+1)
! end if

```

```

// C/C++
float *AR; // malloc( sizeof(float)*N )
float *AC; // malloc( sizeof(float)*N )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PACK_FORMAT );
...

```

```

// on input: R{k} = AR[k]
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
// on output: Z{k} = re + I*im, where
// if (k == 0) {
//     re = AC[0];
//     im = 0;
// } else if (k == N-k) {
//     re = AC[2*k-1];
//     im = 0;
// } else if (k <= N/2) {
//     re = AC[2*k-1];
//     im = AC[2*k-0];
// } else {
//     re = AC[2*(N-k)-1];
//     im = -AC[2*(N-k)-0];
// }

```

DFTI_PACK_FORMAT for Two-dimensional Transforms

The following figure illustrates the storage of a 2D M -by- N conjugate-even sequence in a real array for the PACK packed format. This format requires an array of size M -by- N . Row-major layout and zero-based indexing are used. Different colors mark logically separate parts of the result.

Storage of a 2D M -by- N Conjugate-even Sequence in a Real Array for the PACK Format

	$n=0$	1	2	3	...	$2L-2$	$2L-1$
$m=0$	$R_{0,0}$	$R_{0,1}$	$I_{0,1}$	$R_{0,2}$...	$I_{0,L-1}$	$R_{0,L}$

1	$R_{1,0}$	$R_{1,1}$	$I_{1,1}$	$R_{1,2}$...	$I_{1,L-1}$	$R_{1,L}$
2	$I_{1,0}$	$R_{2,1}$	$I_{2,1}$	$R_{2,2}$...	$I_{2,L-1}$	$I_{2,L}$
3	$R_{2,0}$	$R_{3,1}$	$I_{3,1}$	$R_{3,2}$...	$I_{3,L-1}$	$R_{3,L}$
...
$2K-2$	$I_{K-1,0}$	$R_{2K-2,1}$	$I_{2K-2,1}$	$R_{2K-2,2}$...	$I_{2K-2,L-1}$	$I_{K-1,L}$
$2K-1$	$R_{K,0}$	$R_{2K-1,1}$	$I_{2K-1,1}$	$R_{2K-1,2}$...	$I_{2K-1,L-1}$	$R_{K,L}$

PACK, $M=2K$, $N=2L$

	$n=0$	1	2	3	...	$2L-2$	$2L-1$	$2L$
$m=0$	$R_{0,0}$	$R_{0,1}$	$I_{0,1}$	$R_{0,2}$...	$I_{0,L-1}$	$R_{0,L}$	$I_{0,L}$

1	$R_{1,0}$	$R_{1,1}$	$I_{1,1}$	$R_{1,2}$...	$I_{1,L-1}$	$R_{1,L}$	$I_{1,L}$
2	$I_{1,0}$	$R_{2,1}$	$I_{2,1}$	$R_{2,2}$...	$I_{2,L-1}$	$I_{2,L}$	$I_{2,L}$
3	$R_{2,0}$	$R_{3,1}$	$I_{3,1}$	$R_{3,2}$...	$I_{3,L-1}$	$R_{3,L}$	$I_{3,L}$
...
$2K-2$	$I_{K-1,0}$	$R_{2K-2,1}$	$I_{2K-2,1}$	$R_{2K-2,2}$...	$I_{2K-2,L-1}$	$R_{2K-2,L}$	$I_{2K-2,L}$
$2K-1$	$R_{K,0}$	$R_{2K-1,1}$	$I_{2K-1,1}$	$R_{2K-1,2}$...	$I_{2K-1,L-1}$	$R_{2K-1,L}$	$I_{2K-1,L}$

PACK, $M=2K$, $N=2L+1$

	$n=0$	1	2	3	...	$2L-2$	$2L-1$
$m=0$	$R_{0,0}$	$R_{0,1}$	$I_{0,1}$	$R_{0,2}$...	$I_{0,L-1}$	$R_{0,L}$

1	$R_{1,0}$	$R_{1,1}$	$I_{1,1}$	$R_{1,2}$...	$I_{1,L-1}$	$R_{1,L}$
2	$I_{1,0}$	$R_{2,1}$	$I_{2,1}$	$R_{2,2}$...	$I_{2,L-1}$	$I_{2,L}$
3	$R_{2,0}$	$R_{3,1}$	$I_{3,1}$	$R_{3,2}$...	$I_{3,L-1}$	$R_{3,L}$
...
$2K-2$	$I_{K-1,0}$	$R_{2K-2,1}$	$I_{2K-2,1}$	$R_{2K-2,2}$...	$I_{2K-2,L-1}$	$I_{K-1,L}$
$2K-1$	$R_{K,0}$	$R_{2K-1,1}$	$I_{2K-1,1}$	$R_{2K-1,2}$...	$I_{2K-1,L-1}$	$R_{K,L}$
$2K$	$I_{K,0}$	$R_{2K,1}$	$I_{2K,1}$	$R_{2K,2}$...	$I_{2K,L-1}$	$I_{K,L}$

PACK, $M=2K+1$, $N=2L$

	$n=0$	1	2	3	...	$2L-2$	$2L-1$	$2L$
$m=0$	$R_{0,0}$	$R_{0,1}$	$I_{0,1}$	$R_{0,2}$...	$I_{0,L-1}$	$R_{0,L}$	$I_{0,L}$

1	$R_{1,0}$	$R_{1,1}$	$I_{1,1}$	$R_{1,2}$...	$I_{1,L-1}$	$R_{1,L}$
2	$I_{1,0}$	$R_{2,1}$	$I_{2,1}$	$R_{2,2}$...	$I_{2,L-1}$	$R_{2,L}$
3	$R_{2,0}$	$R_{3,1}$	$I_{3,1}$	$R_{3,2}$...	$I_{3,L-1}$	$R_{3,L}$
...
$2K-2$	$I_{K-1,0}$	$R_{2K-2,1}$	$I_{2K-2,1}$	$R_{2K-2,2}$...	$I_{2K-2,L-1}$	$R_{2K-2,L}$
$2K-1$	$R_{K,0}$	$R_{2K-1,1}$	$I_{2K-1,1}$	$R_{2K-1,2}$...	$I_{2K-1,L-1}$	$R_{2K-1,L}$
$2K$	$I_{K,0}$	$R_{2K,1}$	$I_{2K,1}$	$R_{2K,2}$...	$I_{2K,L-1}$	$R_{2K,L}$

PACK, $M=2K+1$, $N=2L+1$

The real and imaginary parts of the complex-valued conjugate-even sequence $Z_{k1,k2}$ are located in a real-valued array AC as illustrated by figure "Storage of a 2D M -by- N Conjugate-even Sequence in a Real Array for the PACK Format" and can be used to reconstruct the whole sequence as follows:

```

! Fortran
real :: AR(N1,N2), AC(N1,N2)
...

```

```

status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PACK_FORMAT )
...
! on input: R{k1,k2} = AR(k1,k2)
status = DftiComputeForward( desc, AR(:,1) AC(:,1) ) ! real-to-complex FFT
! on output: Z{k1,k2} = cmplx( re, im ), where
! if (k2 == 1) then
!   if (k1 == 1) then
!     re = AC(1,1)
!     im = 0
!   else if (k1-1 == N1-k1+1) then
!     re = AC(2*(k1-1),1)
!     im = 0
!   else if (k1 <= N1/2+1) then
!     re = AC(2*(k1-1)+0,1)
!     im = AC(2*(k1-1)+1,1)
!   else
!     re = AC(2*(N1-k1+1)+0,1)
!     im = -AC(2*(N1-k1+1)+1,1)
!   end if
! else if (k1 == 1) then
!   if (k2-1 == N2-k2+1) then
!     re = AC(1,N2)
!     im = 0
!   else if (k2 <= N2/2+1) then
!     re = AC(1,2*(k2-1)+0)
!     im = AC(1,2*(k2-1)+1)
!   else
!     re = AC(1,2*(N2-k2+1)+0)
!     im = -AC(1,2*(N2-k2+1)+1)
!   endif
! else if (k1-1 == N1-k1+1) then
!   if (k2-1 == N2-k2+1) then
!     re = AC(N1,N2)
!     im = 0
!   else if (k2 <= N2/2+1) then
!     re = AC(N1,2*(k2-1)+0)
!     im = AC(N1,2*(k2-1)+1)
!   else
!     re = AC(N1,2*(N2-k2+1)+0)
!     im = -AC(N1,2*(N2-k2+1)+1)
!   end if
! else if (k1 <= N1/2+1) then
!   re = AC(2*(k1-1)+0,1+k2-1)
!   im = AC(2*(k1-1)+1,1+k2-1)
! else
!   re = AC(2*(N1-k1+1)+0,1+N2-k2+1)
!   im = -AC(2*(N1-k1+1)+1,1+N2-k2+1)
! end if

```

```

// C/C++
float *AR; // malloc( sizeof(float)*N1*N2 )
float *AC; // malloc( sizeof(float)*N1*N2 )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PACK_FORMAT );
...
// on input: R{k1,k2} = AR[N2*k1 + k2]
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
// on output: Z{k1,k2} = re + I*im, where
// if (k1==0) {

```

```

//      if (k2 == 0) {
//          re = AC[0];
//          im = 0;
//      } else if (k2 == N2-k2) {
//          re = AC[2*k2-1];
//          im = 0;
//      } else if (k2 <= N2/2) {
//          re = AC[2*k2-1];
//          im = AC[2*k2-0];
//      } else {
//          re = AC[2*(N2-k2)-1];
//          im = -AC[2*(N2-k2)-0];
//      }
//  } else if (k2==0) {
//      if (k1 == N1-k1) {
//          re = AC[(N1-1)*N2];
//          im = 0;
//      } else if (k1 <= N1/2) {
//          re = AC[(2*k1-1)*N2];
//          im = AC[(2*k1-0)*N2];
//      } else {
//          re = AC[(2*(N1-k1)-1)*N2];
//          im = -AC[(2*(N1-k1)-0)*N2];
//      }
//  } else if (k2 == N2-k2) {
//      if (k1 == N1-k1) {
//          re = AC[N1*N2 - 1];
//          im = 0;
//      } else if (k1 <= N1/2) {
//          re = AC[(2*k1 - 1)*N2 + N2-1];
//          im = AC[(2*k1 - 0)*N2 + N2-1];
//      } else {
//          re = AC[(2*(N1-k1) - 1)*N2 + N2-1];
//          im = -AC[(2*(N1-k1) - 0)*N2 + N2-1];
//      }
//  } else if (k2 <= N2/2) {
//      re = AC[k1*N2+2*k2-1];
//      im = AC[k1*N2+2*k2-0];
//  } else {
//      re = AC[(N1-k1)*N2+2*(N2-k2) - 1];
//      im = -AC[(N1-k1)*N2+2*(N2-k2) - 0];
//  }
// }
```

DFTI_PERM_FORMAT for One-dimensional Transforms

The real and imaginary parts of the complex-valued conjugate-even sequence Z_k are located in real-valued array `AC` as illustrated by figure "[Storage of a 1D Size- \$N\$ Conjugate-even Sequence in a Real Array](#)" and can be used to reconstruct the whole conjugate-even sequence as follows:

```

! Fortran
real :: AR(N), AC(N)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PERM_FORMAT )
...
! on input: R{k} = AR(k)
status = DftiComputeForward( desc, AR, AC ) ! real-to-complex FFT
! on output: Z{k} = cmplx( re, im ), where
! if (k == 1) then
!   re = AC(1)
!   im = 0
! else if (k-1 == N-k+1) then
!   re = AC(2)
```

```
!     im = 0
! else if (k <= N/2+1) then
!   re = AC(1+2*(k-1)+0-mod(N,2))
!   im = AC(1+2*(k-1)+1-mod(N,2))
! else
!   re = AC(1+2*(N-k+1)+0-mod(N,2))
!   im = -AC(1+2*(N-k+1)+1-mod(N,2))
! end if
```

```
// C/C++
float *AR; // malloc( sizeof(float)*N )
float *AC; // malloc( sizeof(float)*N )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PERM_FORMAT );
...
// on input: R{k} = AR[k]
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
// on output: Z{k} = re + I*im, where
// if (k == 0) {
//   re = AC[0];
//   im = 0;
// } else if (k == N-k) {
//   re = AC[1];
//   im = 0;
// } else if (k <= N/2) {
//   re = AC[2*k+0 - N%2];
//   im = AC[2*k+1 - N%2];
// } else {
//   re = AC[2*(N-k)+0 - N%2];
//   im = -AC[2*(N-k)+1 - N%2];
// }
```

DFTI_PERM_FORMAT for Two-dimensional Transforms

The following figure illustrates the storage of a 2D M -by- N conjugate-even sequence in a real array for the PERM packed format. This format requires an array of size M -by- N . Row-major layout and zero-based indexing are used. Different colors mark logically separate parts of the result.

Storage of a 2D M -by- N Conjugate-Even Sequence in a Real Array for the PERM Format

	$n=0$	1	2	3	...	$2L-2$	$2L-1$		$n=0$	1	2	3	...	$2L-2$	$2L-1$	$2L$
$m=0$	$R_{0,0}$	$R_{0,L}$	$R_{0,1}$	$I_{0,1}$...	$R_{0,L-1}$	$I_{0,L-1}$		$R_{0,0}$	$R_{0,1}$	$I_{0,1}$	$R_{0,2}$...	$I_{0,L-1}$	$R_{0,L}$	$I_{0,L}$
1	$R_{1,0}$	$R_{1,L}$	$R_{1,1}$	$I_{1,1}$...	$R_{1,L-1}$	$I_{1,L-1}$		$R_{1,0}$	$R_{1,1}$	$I_{1,1}$	$R_{1,2}$...	$I_{1,L-1}$	$R_{1,L}$	$I_{1,L}$
2	$R_{1,0}$	$R_{2,L}$	$R_{2,1}$	$I_{2,1}$...	$R_{2,L-1}$	$I_{2,L-1}$		$R_{1,0}$	$R_{2,1}$	$I_{2,1}$	$R_{2,2}$...	$I_{2,L-1}$	$R_{2,L}$	$I_{2,L}$
3	$I_{1,0}$	$L_{2,L}$	$R_{3,1}$	$I_{3,1}$...	$R_{3,L-1}$	$I_{3,L-1}$		$I_{1,0}$	$R_{3,1}$	$I_{3,1}$	$R_{3,2}$...	$I_{3,L-1}$	$R_{3,L}$	$I_{3,L}$
...
$2K-2$	$R_{K-1,0}$	$R_{K-1,L}$	$R_{2K-2,1}$	$I_{2K-2,1}$...	$R_{2K-2,L-1}$	$I_{2K-2,L-1}$		$R_{K-1,0}$	$R_{2K-2,1}$	$I_{2K-2,1}$	$R_{2K-2,2}$...	$I_{2K-2,L-1}$	$R_{2K-2,L}$	$I_{2K-2,L}$
$2K-1$	$I_{K-1,0}$	$I_{K-1,L}$	$R_{2K-1,1}$	$I_{2K-1,1}$...	$R_{2K-1,L-1}$	$I_{2K-1,L-1}$		$I_{K-1,0}$	$R_{2K-1,1}$	$I_{2K-1,1}$	$R_{2K-1,2}$...	$I_{2K-1,L-1}$	$R_{2K-1,L}$	$I_{2K-1,L}$

PERM, $M=2K$, $N=2L$								PERM, $M=2K$, $N=2L+1$								
	$n=0$	1	2	3	...	$2L-2$	$2L-1$		$n=0$	1	2	3	...	$2L-2$	$2L-1$	$2L$
$m=0$	$R_{0,0}$	$R_{0,L}$	$R_{0,1}$	$I_{0,1}$...	$R_{0,L-1}$	$I_{0,L-1}$		$R_{0,0}$	$R_{0,1}$	$I_{0,1}$	$R_{0,2}$...	$I_{0,L-1}$	$R_{0,L}$	$I_{0,L}$
1	$R_{1,0}$	$R_{1,L}$	$R_{1,1}$	$I_{1,1}$...	$R_{1,L-1}$	$I_{1,L-1}$		$R_{1,0}$	$R_{1,1}$	$I_{1,1}$	$R_{1,2}$...	$I_{1,L-1}$	$R_{1,L}$	$I_{1,L}$
2	$I_{1,0}$	$I_{1,L}$	$R_{2,1}$	$I_{2,1}$...	$R_{2,L-1}$	$I_{2,L-1}$		$I_{1,0}$	$R_{2,1}$	$I_{2,1}$	$R_{2,2}$...	$I_{2,L-1}$	$R_{2,L}$	$I_{2,L}$
3	$R_{2,0}$	$R_{2,L}$	$R_{3,1}$	$I_{3,1}$...	$R_{3,L-1}$	$I_{3,L-1}$		$R_{2,0}$	$R_{3,1}$	$I_{3,1}$	$R_{3,2}$...	$I_{3,L-1}$	$R_{3,L}$	$I_{3,L}$
...
$2K-2$	$I_{K-1,0}$	$I_{K-1,L}$	$R_{2K-2,1}$	$I_{2K-2,1}$...	$R_{2K-2,L-1}$	$I_{2K-2,L-1}$		$I_{K-1,0}$	$R_{2K-2,1}$	$I_{2K-2,1}$	$R_{2K-2,2}$...	$I_{2K-2,L-1}$	$R_{2K-2,L}$	$I_{2K-2,L}$
$2K-1$	$R_{K,0}$	$R_{K,L}$	$R_{2K-1,1}$	$I_{2K-1,1}$...	$R_{2K-1,L-1}$	$I_{2K-1,L-1}$		$R_{K,0}$	$R_{2K-1,1}$	$I_{2K-1,1}$	$R_{2K-1,2}$...	$I_{2K-1,L-1}$	$R_{2K-1,L}$	$I_{2K-1,L}$
$2K$	$I_{K,0}$	$I_{K,L}$	$R_{2K,1}$	$I_{2K,1}$...	$I_{2K,L-1}$	$I_{2K,L-1}$		$I_{K,0}$	$R_{2K,1}$	$I_{2K,1}$	$R_{2K,2}$...	$I_{2K,L-1}$	$R_{2K,L}$	$I_{2K,L}$

PERM, $M=2K+1$, $N=2L$								PERM, $M=2K+1$, $N=2L+1$								
	$n=0$	1	2	3	...	$2L-2$	$2L-1$		$n=0$	1	2	3	...	$2L-2$	$2L-1$	$2L$
$m=0$	$R_{0,0}$	$R_{0,L}$	$R_{0,1}$	$I_{0,1}$...	$R_{0,L-1}$	$I_{0,L-1}$		$R_{0,0}$	$R_{0,1}$	$I_{0,1}$	$R_{0,2}$...	$I_{0,L-1}$	$R_{0,L}$	$I_{0,L}$
1	$R_{1,0}$	$R_{1,L}$	$R_{1,1}$	$I_{1,1}$...	$R_{1,L-1}$	$I_{1,L-1}$		$R_{1,0}$	$R_{1,1}$	$I_{1,1}$	$R_{1,2}$...	$I_{1,L-1}$	$R_{1,L}$	$I_{1,L}$
2	$I_{1,0}$	$I_{1,L}$	$R_{2,1}$	$I_{2,1}$...	$R_{2,L-1}$	$I_{2,L-1}$		$I_{1,0}$	$R_{2,1}$	$I_{2,1}$	$R_{2,2}$...	$I_{2,L-1}$	$R_{2,L}$	$I_{2,L}$
3	$R_{2,0}$	$R_{2,L}$	$R_{3,1}$	$I_{3,1}$...	$R_{3,L-1}$	$I_{3,L-1}$		$R_{2,0}$	$R_{3,1}$	$I_{3,1}$	$R_{3,2}$...	$I_{3,L-1}$	$R_{3,L}$	$I_{3,L}$
...
$2K-2$	$I_{K-1,0}$	$I_{K-1,L}$	$R_{2K-2,1}$	$I_{2K-2,1}$...	$R_{2K-2,L-1}$	$I_{2K-2,L-1}$		$I_{K-1,0}$	$R_{2K-2,1}$	$I_{2K-2,1}$	$R_{2K-2,2}$...	$I_{2K-2,L-1}$	$R_{2K-2,L}$	$I_{2K-2,L}$
$2K-1$	$R_{K,0}$	$R_{K,L}$	$R_{2K-1,1}$	$I_{2K-1,1}$...	$R_{2K-1,L-1}$	$I_{2K-1,L-1}$		$R_{K,0}$	$R_{2K-1,1}$	$I_{2K-1,1}$	$R_{2K-1,2}$...	$I_{2K-1,L-1}$	$R_{2K-1,L}$	$I_{2K-1,L}$
$2K$	$I_{K,0}$	$I_{K,L}$	$R_{2K,1}$	$I_{2K,1}$...	$I_{2K,L-1}$	$I_{2K,L-1}$		$I_{K,0}$	$R_{2K,1}$	$I_{2K,1}$	$R_{2K,2}$...	$I_{2K,L-1}$	$R_{2K,L}$	$I_{2K,L}$

The real and imaginary parts of the complex-valued conjugate-even sequence $Z_{k1,k2}$ are located in real-valued array AC as illustrated by figure "Storage of a 2D M -by- N Conjugate-even Sequence in a Real Array for the PERM Format" and can be used to reconstruct the whole sequence as follows:

```

! Fortran
real :: AR(N1,N2), AC(N1,N2)
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PERM_FORMAT )
...
! on input: R{k1,k2} = AR(k1,k2)
status = DftiComputeForward( desc, AR(:,1) AC(:,1) ) ! real-to-complex FFT
! on output: Z{k1,k2} = cmplx( re, im ), where
! if (k2 == 1) then
!   if (k1 == 1) then
!     re = AC(1,1)
!     im = 0
!   else if (k1-1 == N1-k1+1) then
!     re = AC(2,1)
!     im = 0
!   else if (k1 <= N1/2+1) then
!     re = AC(1+2*(k1-1)+0 - mod(N1,2),1)
!     im = AC(1+2*(k1-1)+1 - mod(N1,2),1)
!   else
!     re = AC(1+2*(N1-k1+1)+0 - mod(N1,2),1)
!     im = -AC(1+2*(N1-k1+1)+1 - mod(N1,2),1)
!
```

```

!    end if
! else if (k1 == 1) then
!     if (k2-1 == N2-k2+1) then
!       re = AC(1,2)
!       im = 0
!     else if (k2 <= N2/2+1) then
!       re = AC(1,1+2*(k2-1)+0 - mod(N2,2))
!       im = AC(1,1+2*(k2-1)+1 - mod(N2,2))
!     else
!       re = AC(1,1+2*(N2-k2+1)+0 - mod(N2,2))
!       im = -AC(1,1+2*(N2-k2+1)+1 - mod(N2,2))
!     endif
! else if (k1-1 == N1-k1+1) then
!   if (k2-1 == N2-k2+1) then
!     re = AC(2,2)
!     im = 0
!   else if (k2 <= N2/2+1) then
!     re = AC(2,1+2*(k2-1)+0-mod(N2,2))
!     im = AC(2,1+2*(k2-1)+1-mod(N2,2))
!   else
!     re = AC(2,1+2*(N2-k2+1)+0-mod(N2,2))
!     im = -AC(2,1+2*(N2-k2+1)+1-mod(N2,2))
!   end if
! else if (k1 <= N1/2+1) then
!   re = AC(1+2*(k1-1)+0-mod(N1,2),1+k2-1)
!   im = AC(1+2*(k1-1)+1-mod(N1,2),1+k2-1)
! else
!   re = AC(1+2*(N1-k1+1)+0-mod(N1,2),1+N2-k2+1)
!   im = -AC(1+2*(N1-k1+1)+1-mod(N1,2),1+N2-k2+1)
! end if

```

```

// C/C++
float *AR; // malloc( sizeof(float)*N1*N2 )
float *AC; // malloc( sizeof(float)*N1*N2 )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PERM_FORMAT );
...
// on input: R{k1,k2} = AR[N2*k1 + k2]
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
// on output: Z{k1,k2} = re + I*im, where
//  if (k1==0) {
//    if (k2 == 0) {
//      re = AC[0];
//      im = 0;
//    } else if (k2 == N2-k2) {
//      re = AC[1];
//      im = 0;
//    } else if (k2 <= N2/2) {
//      re = AC[2*k2+0 - N2%2];
//      im = AC[2*k2+1 - N2%2];
//    } else {
//      re = AC[2*(N2-k2)+0 - N2%2];
//      im = -AC[2*(N2-k2)+1 - N2%2];
//    }
//  }
//  else if (k2==0) {
//    if (k1 == N1-k1) {
//      re = AC[N2];
//      im = 0;
//    } else if (k1 <= N1/2) {

```

```

//      re = AC[(2*k1+0 - N1%2)*N2];
//      im = AC[(2*k1+1 - N1%2)*N2];
// } else {
//     re = AC[(2*(N1-k1)+0 - N1%2)*N2];
//     im = -AC[(2*(N1-k1)+1 - N1%2)*N2];
// }
// else if (k2 == N2-k2) {
//     if (k1 == N1-k1) {
//         re = AC[N2 + 1];
//         im = 0;
//     } else if (k1 <= N1/2) {
//         re = AC[(2*k1+0 - N1%2)*N2 + 1];
//         im = AC[(2*k1+1 - N1%2)*N2 + 1];
//     } else {
//         re = AC[(2*(N1-k1)+0 - N1%2)*N2 + 1];
//         im = -AC[(2*(N1-k1)+1 - N1%2)*N2 + 1];
//     }
// } else if (k2 <= N2/2) {
//     re = AC[k1*N2+2*k2+0 - N2%2];
//     im = AC[k1*N2+2*k2+1 - N2%2];
// } else {
//     re = AC[(N1-k1)*N2+2*(N2-k2)+0 - N2%2];
//     im = -AC[(N1-k1)*N2+2*(N2-k2)+1 - N2%2];
// }

```

To better understand packed formats for two-dimensional transforms, refer to the following examples in your Intel MKL directory:

C:

`./examples/dftc/source/config_conjugate_even_storage.c`

Fortran:

`./examples/dftf/source/config_conjugate_even_storage.f90`

See Also

[DftiSetValue](#)

DFTI_WORKSPACE

The computation step for some FFT algorithms requires a scratch space for permutation or other purposes. To manage the use of the auxiliary storage, Intel MKL enables you to set the configuration parameter `DFTI_WORKSPACE` with the following values:

`DFTI_ALLOW` (default) Permits the use of the auxiliary storage.

`DFTI_AVOID` Instructs Intel MKL to avoid using the auxiliary storage if possible.

See Also

[DftiSetValue](#)

DFTI_COMMIT_STATUS

The `DFTI_COMMIT_STATUS` configuration parameter indicates whether the descriptor is ready for computation. The parameter has two possible values:

`DFTI_UNCOMMITTED` Default value, set after a successful call of `DftiCreateDescriptor`.

`DFTI_COMMITTED` The value after a successful call to `DftiCommitDescriptor`.

A computation function called with an uncommitted descriptor returns an error.

You cannot directly set this configuration parameter in a call to `DftiSetValue`, but a change in the configuration of a committed descriptor may change the commit status of the descriptor to `DFTI_UNCOMMITTED`.

See Also

[DftiCreateDescriptor](#)
[DftiCommitDescriptor](#)
[DftiSetValue](#)

DFTI_ORDERING

Some FFT algorithms apply an explicit permutation stage that is time consuming [4]. The exclusion of this step is similar to applying an FFT to input data whose order is scrambled, or allowing a scrambled order of the FFT results. In applications such as convolution and power spectrum calculation, the order of result or data is unimportant and thus using scrambled data is acceptable if it leads to better performance. The following options are available in Intel MKL:

- `DFTI_ORDERED`: Forward transform data ordered, backward transform data ordered (default option).
- `DFTI_BACKWARD_SCRAMBLED`: Forward transform data ordered, backward transform data scrambled.

Table "Scrambled Order Transform" tabulates the effect of this configuration setting.

Scrambled Order Transform

	<code>DftiComputeForward</code>	<code>DftiComputeBackward</code>
<code>DFTI_ORDERING</code>	Input → Output	Input → Output
<code>DFTI_ORDERED</code>	ordered → ordered	ordered → ordered
<code>DFTI_BACKWARD_SCRAMBLED</code>	ordered → scrambled	scrambled → ordered

NOTE

The word "scrambled" in this table means "permit scrambled order if possible". In some situations permitting out-of-order data gives no performance advantage and an implementation may choose to ignore the suggestion.

See Also

[DftiSetValue](#)

Descriptor Manipulation Functions

This category contains the following functions: create a descriptor, commit a descriptor, copy a descriptor, and free a descriptor.

DftiCreateDescriptor

Allocates the descriptor data structure and initializes it with default configuration values.

Syntax

Fortran:

```
status = DftiCreateDescriptor( desc_handle, precision, forward_domain, dimension,
length )
```

C:

```
status = DftiCreateDescriptor(&desc_handle, precision, forward_domain, dimension,
length);
```

Include Files

- Fortran 90: mkl_dfti.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>precision</i>	FORTRAN: INTEGER C: enum	Precision of the transform: DFTI_SINGLE or DFTI_DOUBLE.
<i>forward_domain</i>	FORTRAN: INTEGER C: enum	Forward domain of the transform: DFTI_COMPLEX or DFTI_REAL.
<i>dimension</i>	FORTRAN: INTEGER C: MKL_LONG	Dimension of the transform.
<i>length</i>	FORTRAN: INTEGER if <i>dimension</i> = 1. Array INTEGER, DIMENSION(*) otherwise. C: MKL_LONG if <i>dimension</i> = 1. Array of type MKL_LONG otherwise.	Length of the transform for a one-dimensional transform. Lengths of each dimension for a multi-dimensional transform.

Output Parameters

Name	Type	Description
<i>desc_handle</i>	FORTRAN: DFTI_DESCRIPTOR C: DFTI_DESCRIPTOR_HANDLE	FFT descriptor.
<i>status</i>	FORTRAN: INTEGER C: MKL_LONG	Function completion status.

Description

This function allocates memory for the descriptor data structure and instantiates it with all the default configuration settings for the precision, forward domain, dimension, and length of the desired transform. Because memory is allocated dynamically, the result is actually a pointer to the created descriptor. This function is slightly different from the "initialization" function that can be found in software packages or libraries that implement more traditional algorithms for computing an FFT. This function does not perform any significant computational work such as computation of twiddle factors. The function [DftiCommitDescriptor](#) does this work after the function [DftiSetValue](#) has set values of all necessary parameters.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```

! Fortran interface.
! Note that the body provided below only illustrates the list of different
! parameters and the types of dummy parameters. You can rely only on the function
! name following keyword INTERFACE. For the precise definition of the
! interface, see the include/mkl_dfti.f90 file in the Intel MKL directory.

INTERFACE DftiCreateDescriptor

  FUNCTION some_actual_function_1d(desc, precision, domain, dim, length)
    INTEGER :: some_actual_function_1d
    ...
    INTEGER, INTENT(IN) :: length
  END FUNCTION some_actual_function_1d

  FUNCTION some_actual_function_md(desc, precision, domain, dim, lengths)
    INTEGER :: some_actual_function_md
    ...
    INTEGER, INTENT(IN), DIMENSION(*) :: lengths
  END FUNCTION some_actual_function_md

  ...

END INTERFACE DftiCreateDescriptor

```

Note that the function is overloaded: the actual parameter for the formal parameter *length* can be a scalar or a rank-one array.

The function is also overloaded with respect to the type of the *precision* parameter in order to provide an option of using a precision-specific function for the generic name. Using more specific functions can reduce the size of statically linked executable for the applications using only single-precision FFTs or only double-precision FFTs. To use specific functions, change the "USE MKL_DFTI" statement in your program unit to one of the following:

```
USE MKL_DFTI, FORGET=>DFTI_SINGLE, DFTI_SINGLE=>DFTI_SINGLE_R
```

```
USE MKL_DFTI, FORGET=>DFTI_DOUBLE, DFTI_DOUBLE=>DFTI_DOUBLE_R
```

where the name "FORGET" can be replaced with any name that is not used in the program unit.

Prototype

```

/* C prototype.
 * Note that the preprocessor definition provided below only illustrates
 * that the actual function called may be determined at compile time.
 * You can rely only on the declaration of the function.
 * For precise definition of the preprocessor macro, see the include/mkl_dfti.h
 * file in the Intel MKL directory.
 */
MKL_LONG DftiCreateDescriptor(DFTI_DESCRIPTOR_HANDLE * pHandle,
    enum DFTI_CONFIG_VALUE precision,
    enum DFTI_CONFIG_VALUE domain,
    MKL_LONG dimension, ... /* length/lengths */ );

#define DftiCreateDescriptor(desc,prec,domain,dim,sizes) \
  ((prec)==DFTI_SINGLE && (dim)==1) ? \

```

```
some_actual_function_s1d((desc),(domain),(MKL_LONG)(sizes)) : \
...
```

Variable *length/lengths* is interpreted as a scalar (`MKL_LONG`) or an array (`MKL_LONG*`), depending on the value of parameter *dimension*. If the value of parameter *precision* is known at compile time, an optimizing compiler retains only the call to the respective specific function, thereby reducing the size of the statically linked application. Avoid direct calls to the specific functions used in the preprocessor macro definition, because their interface may change in future releases of the library. If the use of the macro is undesirable, you can safely undefine it after inclusion of the Intel MKL FFT header file, as follows:

```
#include "mkl_dfti.h"
#undef DftiCreateDescriptor
```

See Also

[DFTI_PRECISION](#) configuration parameter

[DFTI_FORWARD_DOMAIN](#) configuration parameter

[DFTI_DIMENSION](#), [DFTI_LENGTHS](#) configuration parameters

[Configuration Parameters](#)

DftiCommitDescriptor

Performs all initialization for the actual FFT computation.

Syntax

Fortran:

```
status = DftiCommitDescriptor( desc_handle )
```

C:

```
status = DftiCommitDescriptor(desc_handle);
```

Include Files

- Fortran 90: `mkl_dfti.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>desc_handle</code>	<code>FORTTRAN: DFTI_DESCRIPTOR</code> <code>C: DFTI_DESCRIPTOR_HANDLE</code>	FFT descriptor.

Output Parameters

Name	Type	Description
<code>desc_handle</code>	<code>FORTTRAN: DFTI_DESCRIPTOR</code> <code>C: DFTI_DESCRIPTOR_HANDLE</code>	Updated FFT descriptor.
<code>status</code>	<code>FORTTRAN: INTEGER</code> <code>C: MKL_LONG</code>	Function completion status.

Description

This function completes initialization of a previously created descriptor, which is required before the descriptor can be used for FFT computations. Typically, committing the descriptor performs all initialization that is required for the actual FFT computation. The initialization done by the function may involve exploring different factorizations of the input length to find the optimal computation method.

If you call the `DftiSetValue` function to change configuration parameters of a committed descriptor (see [Descriptor Configuration Functions](#)), you must re-commit the descriptor before invoking a computation function. Typically, a committal function call is immediately followed by a computation function call (see [FFT Computation Functions](#)).

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```

! Fortran interface
INTERFACE DftiCommitDescriptor
!Note that the body provided here is to illustrate the different
!argument list and types of dummy arguments. The interface
!does not guarantee what the actual function names are.
!Users can only rely on the function name following the
!keyword INTERFACE
  FUNCTION some_actual_function_1 ( Desc_Handle )
    INTEGER :: some_actual_function_1
    TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
  END FUNCTION some_actual_function_1
END INTERFACE DftiCommitDescriptor

```

Prototype

```

/* C prototype */
MKL_LONG DftiCommitDescriptor( DFTI_DESCRIPTOR_HANDLE );

```

DftiFreeDescriptor

Frees the memory allocated for a descriptor.

Syntax

Fortran:

```
status = DftiFreeDescriptor( desc_handle )
```

C:

```
status = DftiFreeDescriptor(&desc_handle);
```

Include Files

- Fortran 90: `mkl_dfti.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>desc_handle</code>	<code>FORTTRAN: DESCRIPTOR_HANDLE</code>	FFT descriptor.

Name	Type	Description
	C: DFTI_DESCRIPTOR_HANDLE	

Output Parameters

Name	Type	Description
desc_handle	FORTRAN: DESCRIPTOR_HANDLE C: DFTI_DESCRIPTOR_HANDLE	Memory for the FFT descriptor is released.
status	FORTRAN: INTEGER C: MKL_LONG	Function completion status.

Description

This function frees all memory allocated for a descriptor.

NOTE

Memory allocation/deallocation inside Intel MKL is managed by Intel MKL memory management software. So, even after successful completion of `FreeDescriptor`, the memory space may continue being allocated for the application because the memory management software sometimes does not return the memory space to the OS, but considers the space free and can reuse it for future memory allocation. See [Example "mkl_free_buffers Usage with FFT Functions"](#) in the description of the service function `FreeBuffers` on how to use Intel MKL memory management software and release memory to the OS.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```
! Fortran interface
INTERFACE DftiFreeDescriptor
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_3( Desc_Handle )
INTEGER :: some_actual_function_3
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
END FUNCTION some_actual_function_3
END INTERFACE DftiFreeDescriptor
```

Prototype

```
/* C prototype */
MKL_LONG DftiFreeDescriptor( DFTI_DESCRIPTOR_HANDLE * );
```

DftiCopyDescriptor

Makes a copy of an existing descriptor.

Syntax

Fortran:

```
status = DftiCopyDescriptor( desc_handle_original, desc_handle_copy )
```

C:

```
status = DftiCopyDescriptor(desc_handle_original, &desc_handle_copy);
```

Include Files

- Fortran 90: mkl_dfti.f90
- C: mkl.h

Input Parameters

Name	Type	Description
desc_handle_original	FORTRAN: DESCRIPTOR_HANDLE C: DFTI_DESCRIPTOR_HANDLE	The FFT descriptor to copy.

Output Parameters

Name	Type	Description
desc_handle_copy	FORTRAN: DESCRIPTOR_HANDLE C: DFTI_DESCRIPTOR_HANDLE	The copy of the FFT descriptor.
status	FORTRAN: INTEGER C: MKL_LONG	Function completion status.

Description

This function makes a copy of an existing descriptor and provides a pointer to it. The purpose is that all information of the original descriptor will be maintained even if the original is destroyed via the free descriptor function `DftiFreeDescriptor`.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```
! Fortran interface
INTERFACE DftiCopyDescriptor
! Note that the body provided here is to illustrate the different
! argument list and types of dummy arguments. The interface
! does not guarantee what the actual function names are.
! Users can only rely on the function name following the
! keyword INTERFACE
FUNCTION some_actual_function_2( Desc_Handle_Original,
Desc_Handle_Copy )
INTEGER :: some_actual_function_2
TYPE(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Original, Desc_Handle_Copy
```

```
END FUNCTION some_actual_function_2
END INTERFACE DftiCopyDescriptor
```

Prototype

```
/* C prototype */
MKL_LONG DftiCopyDescriptor( DFTI_DESCRIPTOR_HANDLE, DFTI_DESCRIPTOR_HANDLE * );
```

Descriptor Configuration Functions

This category contains the following functions: the value setting function `DftiSetValue` sets one particular configuration parameter to an appropriate value, and the value getting function `DftiGetValue` reads the value of one particular configuration parameter. While all configuration parameters are readable, you cannot set a few of them. Some of these contain fixed information of a particular implementation such as version number, or dynamic information, which is derived by the implementation during execution of one of the functions. See [Configuration Settings](#) for details.

DftiSetValue

Sets one particular configuration parameter with the specified configuration value.

Syntax

Fortran:

```
status = DftiSetValue( desc_handle, config_param, config_val )
```

C:

```
status = DftiSetValue(desc_handle, config_param, config_val);
```

Include Files

- Fortran 90: `mkl_dfti.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>desc_handle</code>	FORTRAN: DFTI_DESCRIPTOR C: DFTI_DESCRIPTOR_HANDLE	FFT descriptor.
<code>config_param</code>	FORTRAN: INTEGER C: enum	Configuration parameter.
<code>config_val</code>	Depends on the configuration parameter.	Configuration value.

Output Parameters

Name	Type	Description
<code>desc_handle</code>	FORTRAN: DFTI_DESCRIPTOR C: DFTI_DESCRIPTOR_HANDLE	Updated FFT descriptor.

Name	Type	Description
<i>status</i>	FORTRAN: INTEGER C: MKL_LONG	Function completion status.

Description

This function sets one particular configuration parameter with the specified configuration value. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see:

- DFTI_PRECISION
- DFTI_FORWARD_DOMAIN
- DFTI_DIMENSION, DFTI_LENGTH
- DFTI_PLACEMENT
- DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE
- DFTI_THREAD_LIMIT
- DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES
- DFTI_NUMBER_OF_TRANSFORMS
- DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE
- DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE
- DFTI_PACKED_FORMAT
- DFTI_WORKSPACE
- DFTI_ORDERING

The `DftiSetValue` function cannot be used to change configuration parameters `DFTI_FORWARD_DOMAIN`, `DFTI_PRECISION`, `DFTI_DIMENSION`, and `DFTI_LENGTHS`. Use the `DftiCreateDescriptor` function to set them.

Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [Configuring and Computing an FFT in Fortran](#) and [Configuring and Computing an FFT in C/C++](#).

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```

! Fortran interface
INTERFACE DftiSetValue
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_6_INTVAL( Desc_Handle, Config_Param, INTVAL )
INTEGER :: some_actual_function_6_INTVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(IN) :: INTVAL
END FUNCTION some_actual_function_6_INTVAL
FUNCTION some_actual_function_6_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
INTEGER :: some_actual_function_6_SGLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL, INTENT(IN) :: SGLVAL

```

```

END FUNCTION some_actual_function_6_SGLVAL
FUNCTION some_actual_function_6_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
INTEGER :: some_actual_function_6_DBLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL (KIND(0D0)), INTENT(IN) :: DBLVAL
END FUNCTION some_actual_function_6_DBLVAL
FUNCTION some_actual_function_6_INTVEC( Desc_Handle, Config_Param, INTVEC )
INTEGER :: some_actual_function_6_INTVEC
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(IN) :: INTVEC(*)
END FUNCTION some_actual_function_6_INTVEC
FUNCTION some_actual_function_6_CHARS( Desc_Handle, Config_Param, CHARS )
INTEGER :: some_actual_function_6_CHARS
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
CHARACTER(*), INTENT(IN) :: CHARS
END FUNCTION some_actual_function_6_CHARS
END INTERFACE DftiSetValue

```

Prototype

```

/* C prototype */
MKL_LONG DftiSetValue( DFTI_DESCRIPTOR_HANDLE, DFTI_CONFIG_PARAM , ... );

```

See Also

Configuration Settings for more information on configuration parameters.

[DftiCreateDescriptor](#)

[DftiGetValue](#)

DftiGetValue

Gets the configuration value of one particular configuration parameter.

Syntax

Fortran:

```
status = DftiGetValue( desc_handle, config_param, config_val )
```

C:

```
status = DftiGetValue(desc_handle, config_param, &config_val);
```

Include Files

- Fortran 90: mkl_dfti.f90
- C: mkl.h

Input Parameters

Name	Type	Description
desc_handle	FORTRAN: DFTI_DESCRIPTOR C: DFTI_DESCRIPTOR_HANDLE	FFT descriptor.

Name	Type	Description
<i>config_param</i>	FORTRAN: INTEGER C: enum	Configuration parameter. See Table "Configuration Parameters" for allowable values of <i>config_param</i> .

Output Parameters

Name	Type	Description
<i>config_val</i>	Depends on the configuration parameter.	Configuration value.
<i>status</i>	FORTRAN: INTEGER C: MKL_LONG	Function completion status.

Description

This function gets the configuration value of one particular configuration parameter. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see:

- DFTI_PRECISION
- DFTI_FORWARD_DOMAIN
- DFTI_DIMENSION, DFTI_LENGTH
- DFTI_PLACEMENT
- DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE
- DFTI_THREAD_LIMIT
- DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES
- DFTI_NUMBER_OF_TRANSFORMS
- DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE
- DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE
- DFTI_PACKED_FORMAT
- DFTI_WORKSPACE
- DFTI_COMMIT_STATUS
- DFTI_ORDERING

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```

! Fortran interface
INTERFACE DftiGetValue
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_7_INTVAL( Desc_Handle, Config_Param, INTVAL )
INTEGER :: some_actual_function_7_INTVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(OUT) :: INTVAL
END FUNCTION DFTI_GET_VALUE_INTVAL

```

```

FUNCTION some_actual_function_7_SGLVAL( Desc_Handle, Config_Param, SGLVAL )
INTEGER :: some_actual_function_7_SGLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL, INTENT(OUT) :: SGLVAL
END FUNCTION some_actual_function_7_SGLVAL
FUNCTION some_actual_function_7_DBLVAL( Desc_Handle, Config_Param, DBLVAL )
INTEGER :: some_actual_function_7_DBLVAL
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
REAL (KIND(0D0)), INTENT(OUT) :: DBLVAL
END FUNCTION some_actual_function_7_DBLVAL
FUNCTION some_actual_function_7_INTVEC( Desc_Handle, Config_Param, INTVEC )
INTEGER :: some_actual_function_7_INTVEC
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, INTENT(OUT) :: INTVEC(*)
END FUNCTION some_actual_function_7_INTVEC
FUNCTION some_actual_function_7_INTPNT( Desc_Handle, Config_Param, INTPNT )
INTEGER :: some_actual_function_7_INTPNT
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
INTEGER, DIMENSION(*), POINTER :: INTPNT
END FUNCTION some_actual_function_7_INTPNT
FUNCTION some_actual_function_7_CHARS( Desc_Handle, Config_Param, CHARS )
INTEGER :: some_actual_function_7_CHARS
Type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle
INTEGER, INTENT(IN) :: Config_Param
CHARACTER(*), INTENT(OUT):: CHARS
END FUNCTION some_actual_function_7_CHARS
END INTERFACE DftiGetValue

```

Prototype

```

/* C prototype */
MKL_LONG DftiGetValue( DFTI_DESCRIPTOR_HANDLE,
    DFTI_CONFIG_PARAM ,
    ... );

```

See Also

[Configuration Settings](#) for more information on configuration parameters.

[DftiSetValue](#)

FFT Computation Functions

This category contains the following functions: compute the forward transform and compute the backward transform.

DftiComputeForward

Computes the forward FFT.

Syntax

Fortran:

```

status = DftiComputeForward( desc_handle, x_inout )
status = DftiComputeForward( desc_handle, x_in, y_out )

```

```
status = DftiComputeForward( desc_handle, xre_inout, xim_inout )
status = DftiComputeForward( desc_handle, xre_in, xim_in, yre_out, yim_out )
```

C:

```
status = DftiComputeForward(desc_handle, x_inout);
status = DftiComputeForward(desc_handle, x_in, y_out);
status = DftiComputeForward(desc_handle, xre_inout, xim_inout);
status = DftiComputeForward(desc_handle, xre_in, xim_in, yre_out, yim_out);
```

Input Parameters

Name	Type	Description
desc_handle	FORTRAN: DFTI_DESCRIPTOR C: DFTI_DESCRIPTOR_HANDLE	FFT descriptor.
x_inout, x_in	FORTRAN: Array REAL(KIND=WP) or COMPLEX(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor. C: Array of type float or double depending on the precision of the transform, specified in the DFTI_PRECISION configuration setting.	Data to be transformed in case of a real forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.
xre_inout, xim_in	FORTRAN: Array yim_in REAL(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor. C: Array of type float or double depending on the precision of the transform.	Real and imaginary parts of the data to be transformed in the case of a complex forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.

The suffix in parameter names corresponds to the value of the configuration parameter `DFTI_PLACEMENT` as follows:

- `_inout` to `DFTI_INPLACE`
- `_in` to `DFTI_NOT_INPLACE`

Output Parameters

Name	Type	Description
y_out	FORTRAN: Array REAL(KIND=WP) or COMPLEX(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	The transformed data in case of a real backward domain, determined by the DFTI_FORWARD_DOMAIN configuration setting.

Name	Type	Description
	<p>C: Array of type float or double depending on the precision of the transform.</p>	
xre_inout, xim_ino	FORTRAN: <code>REAL(KIND=WP), DIMENSION(*)</code> , where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Real and imaginary parts of the transformed data in the case of a complex backward domain, determined by the <code>DFTI_FORWARD_DOMAIN</code> configuration setting.
	<p>C: Array of type float or double depending on the precision of the transform.</p>	
status	FORTRAN: INTEGER C: MKL_LONG	Function completion status.

The suffix in parameter names corresponds to the value of the configuration parameter `DFTI_PLACEMENT` as follows:

- `_inout` to `DFTI_INPLACE`
- `_out` to `DFTI_NOT_INPLACE`

Include Files

- Fortran 90: `mkl_dfti.f90`
- C: `mkl.h`

Description

The `DftiComputeForward` function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, this function computes the forward FFT, that is, the `transform` with the minus sign in the exponent, $\delta = -1$.

The `DFTI_COMPLEX_STORAGE`, `DFTI_REAL_STORAGE`, and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters define the layout of the input and output data and must be properly set in a call to the `DftiSetValue` function.

The FFT descriptor must be properly configured prior to the function call. Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [Configuring and Computing an FFT in Fortran](#) and [Configuring and Computing an FFT in C/C++](#).

The number and types of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by variable parameters in C and the generic interface in Fortran. The generic Fortran interface to the computation functions is based on a set of specific functions. These functions can check for inconsistency between the required and actual number of parameters. However, the specific functions disregard the type of the actual parameters and instead use the interpretation defined in the descriptor by configuration parameters `DFTI_FORWARD_DOMAIN`, `DFTI_INPUT_STRIDES`, `DFTI_INPUT_DISTANCE`, and so on.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```

! Fortran interface
! Note that the body provided below only illustrates the list of different
! parameters and the types of dummy parameters. You can rely only on the function

```

```

! name following keyword INTERFACE. For the precise definition of the
! interface, see the include/mkl_dfti.f90 file in the Intel MKL directory.
INTERFACE DftiComputeForward

FUNCTION some_actual_function_1(desc,sSrcDst)
  INTEGER some_actual_function_1
  REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDst
  ...
END FUNCTION some_actual_function_1

FUNCTION some_actual_function_2(desc,cSrcDst)
  INTEGER some_actual_function_2
  COMPLEX(8), INTENT(INOUT), DIMENSION(*) :: cSrcDst
  ...
END FUNCTION some_actual_function_2

FUNCTION some_actual_function_3(desc,sSrcDstRe,sSrcDstIm)
  INTEGER some_actual_function_3
  REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstRe
  REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstIm
  ...
END FUNCTION some_actual_function_3
...
END INTERFACE DftiComputeForward

```

The Fortran interface requires that the data parameters have the type of assumed-size rank-1 array, even for multidimensional transforms. The implementations of the FFT interface require the data stored linearly in memory with a regular stride pattern capable of describing multidimensional array layout (see also [3] and the more detailed discussion in [DFTI_INPUT_STRIDES](#), [DFTI_OUTPUT_STRIDES](#)), and the function requires that the data parameters refer to the first element of the data. Consequently, the data arrays should be specified with the `DIMENSION(*)` attribute and the storage associated with the actual multidimensional arrays via the `EQUIVALENCE` statement.

Prototype

```

/* C prototype */
MKL_LONG DftiComputeForward( DFTI_DESCRIPTOR_HANDLE, void*, ... );

```

See Also

[Configuration Settings](#)
[DFTI_FORWARD_DOMAIN](#)
[DFTI_PLACEMENT](#)
[DFTI_PACKED_FORMAT](#)
[DFTI_COMPLEX_STORAGE](#), [DFTI_REAL_STORAGE](#), [DFTI_CONJUGATE_EVEN_STORAGE](#)
[DFTI_DIMENSION](#), [DFTI_LENGTHS](#)
[DFTI_INPUT_DISTANCE](#), [DFTI_OUTPUT_DISTANCE](#)
[DFTI_INPUT_STRIDES](#), [DFTI_OUTPUT_STRIDES](#)
[DftiComputeBackward](#)
[DftiSetValue](#)

DftiComputeBackward

Computes the backward FFT.

Syntax

Fortran:

```

status = DftiComputeBackward( desc_handle, x_inout )
status = DftiComputeBackward( desc_handle, y_in, x_out )
status = DftiComputeBackward( desc_handle, xre_inout, xim_inout )
status = DftiComputeBackward( desc_handle, yre_in, yim_in, xre_out, xim_out )

```

C:

```

status = DftiComputeBackward(desc_handle, x_inout);
status = DftiComputeBackward(desc_handle, y_in, x_out);
status = DftiComputeBackward(desc_handle, xre_inout, xim_inout);
status = DftiComputeBackward(desc_handle, yre_in, yim_in, xre_out, xim_out);

```

Input Parameters

Name	Type	Description
desc_handle	FORTRAN: DFTI_DESCRIPTOR C: DFTI_DESCRIPTOR_HANDLE	FFT descriptor.
x_inout, y_in	FORTRAN: Array REAL(KIND=WP) or COMPLEX(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor. C: Array of type float or double depending on the precision of the transform, specified in the DFTI_PRECISION configuration setting.	Data to be transformed in case of a real backward domain, determined by the DFTI_FORWARD_DOMAIN configuration setting.
xre_inout, xim_inout, yim_in	FORTRAN: Array yim_in REAL(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor. C: Array of type float or double depending on the precision of the transform.	Real and imaginary parts of the data to be transformed in the case of a complex backward domain, determined by the DFTI_FORWARD_DOMAIN configuration setting.

The suffix in parameter names corresponds to the value of the configuration parameter `DFTI_PLACEMENT` as follows:

- `_inout` to `DFTI_INPLACE`
- `_in` to `DFTI_NOT_INPLACE`

Output Parameters

Name	Type	Description
x_out	FORTRAN: Array REAL(KIND=WP) or COMPLEX(KIND=WP), DIMENSION(*), where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	The transformed data in case of a real forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.

Name	Type	Description
	C: Array of type float or double depending on the precision of the transform.	
xre_inout, xim_ino	FORTRAN: <code>REAL(KIND=WP), DIMENSION(*)</code> , where type and working precision WP must be consistent with the forward domain and precision specified in the descriptor.	Real and imaginary parts of the transformed data in the case of a complex forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.
	C: Array of type float or double depending on the precision of the transform.	
status	FORTRAN: INTEGER C: MKL_LONG	Function completion status.

The suffix in parameter names corresponds to the value of the configuration parameter DFTI_PLACEMENT as follows:

- `_inout` to DFTI_INPLACE
- `_out` to DFTI_NOT_INPLACE

Include Files

- Fortran 90: `mkl_dfti.f90`
- C: `mkl.h`

Description

The function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, the `DftiComputeBackward` function computes the inverse FFT, that is, the `transform` with the plus sign in the exponent, $\delta = +1$.

The `DFTI_COMPLEX_STORAGE`, `DFTI_REAL_STORAGE`, and `DFTI_CONJUGATE_EVEN_STORAGE` configuration parameters define the layout of the input and output data and must be properly set in a call to the `DftiSetValue` function.

The FFT descriptor must be properly configured prior to the function call. Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in [Configuring and Computing an FFT in Fortran](#) and [Configuring and Computing an FFT in C/C++](#).

The number and types of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by variable parameters in C and the generic interface in Fortran. The generic Fortran interface to the computation functions is based on a set of specific functions. These functions can check for inconsistency between the required and actual number of parameters. However, the specific functions disregard the type of the actual parameters and instead use the interpretation defined in the descriptor by configuration parameters `DFTI_FORWARD_DOMAIN`, `DFTI_INPUT_STRIDES`, `DFTI_INPUT_DISTANCE`, and so on.

The function returns zero when it completes successfully. See [Status Checking Functions](#) for more information on the returned status.

Interface

```
! Fortran interface
! Note that the body provided below only illustrates the list of different
! parameters and the types of dummy parameters. You can rely only on the function
```

```

! name following keyword INTERFACE. For the precise definition of the
! interface, see the include/mkl_dfti.f90 file in the Intel MKL directory.
INTERFACE DftiComputeBackward

  FUNCTION some_actual_function_1(desc,sSrcDst)
    INTEGER some_actual_function_1
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDst
    ...
  END FUNCTION some_actual_function_1

  FUNCTION some_actual_function_2(desc,cSrcDst)
    INTEGER some_actual_function_2
    COMPLEX(8), INTENT(INOUT), DIMENSION(*) :: cSrcDst
    ...
  END FUNCTION some_actual_function_2

  FUNCTION some_actual_function_3(desc,sSrcDstRe,sSrcDstIm)
    INTEGER some_actual_function_3
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstRe
    REAL(4), INTENT(INOUT), DIMENSION(*) :: sSrcDstIm
    ...
  END FUNCTION some_actual_function_3

  ...
END INTERFACE DftiComputeBackward

```

The Fortran interface requires that the data parameters have the type of assumed-size rank-1 array, even for multidimensional transforms. The implementations of the FFT interface require the data stored linearly in memory with a regular stride pattern capable of describing multidimensional array layout (see also [3] and the more detailed discussion in [DFTI_INPUT_STRIDES](#), [DFTI_OUTPUT_STRIDES](#)), and the function requires that the data parameters refer to the first element of the data. Consequently, the data arrays should be specified with the `DIMENSION(*)` attribute and the storage associated with the actual multidimensional arrays via the `EQUIVALENCE` statement.

Prototype

```

/* C prototype */
MKL_LONG DftiComputeBackward( DFTI_DESCRIPTOR_HANDLE, void *, ... );

```

See Also

[Configuration Settings](#)

[DFTI_FORWARD_DOMAIN](#)

[DFTI_PLACEMENT](#)

[DFTI_PACKED_FORMAT](#)

[DFTI_COMPLEX_STORAGE](#), [DFTI_REAL_STORAGE](#), [DFTI_CONJUGATE_EVEN_STORAGE](#)

[DFTI_DIMENSION](#), [DFTI_LENGTHS](#)

[DFTI_INPUT_DISTANCE](#), [DFTI_OUTPUT_DISTANCE](#)

[DFTI_INPUT_STRIDES](#), [DFTI_OUTPUT_STRIDES](#)

[DftiComputeForward](#)

[DftiSetValue](#)

Configuring and Computing an FFT in Fortran

The table below summarizes information on configuring and computing an FFT in Fortran for all kinds of transforms and possible combinations of input and output domains.

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
Complex-to-complex, in-place, forward or backward	Interleaved complex numbers	Interleaved complex numbers	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_COMPLEX, <dimension>, <sizes>) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT status = DftiComputeForward(hand, X_inout) ! or backward FFT status = DftiComputeBackward(hand, X_inout) </pre>
Complex-to-complex, out-of-place, forward or backward	Interleaved complex numbers	Interleaved complex numbers	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_COMPLEX, <dimension>, <sizes>) status = DftiSetValue(hand, DFTI_PLACEMENT, & DFTI_NOT_INPLACE) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT status = DftiComputeForward(hand, X_in, Y_out) ! or backward FFT status = DftiComputeBackward(hand, X_in, Y_out) </pre>
Complex-to-complex, in-place, forward or backward	Split-complex numbers	Split-complex numbers	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_COMPLEX, <dimension>, <sizes>) status = DftiSetValue(hand, & DFTI_COMPLEX_STORAGE, DFTI_REAL_REAL) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT status = DftiComputeForward(hand, Xre_inout, & Xim_inout) ! or backward FFT status = DftiComputeBackward(hand, Xre_inout, & Xim_inout) </pre>
Complex-to-complex, out-of-place, forward or backward	Split-complex numbers	Split-complex numbers	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_COMPLEX, <dimension>, <sizes>) status = DftiSetValue(hand, & DFTI_COMPLEX_STORAGE, DFTI_REAL_REAL) status = DftiSetValue(hand, & DFTI_PLACEMENT, DFTI_NOT_INPLACE) status = DftiCommitDescriptor(hand) ! Compute an FFT ! forward FFT status = DftiComputeForward(hand, Xre_in, & Xim_in, Yre_out, Yim_out) ! or backward FFT status = DftiComputeBackward(hand, Xre_in, & Xim_in, Yre_out, Yim_out) </pre>

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
Real-to-complex, in-place, forward	Real numbers	Numbers in the CCE format	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_REAL, <dimension>, <sizes>) status = DftiSetValue(hand, & DFTI_CONJUGATE_EVEN_STORAGE, & DFTI_COMPLEX_COMPLEX) status = DftiSetValue(hand, DFTI_PACKED_FORMAT, & DFTI_CCE_FORMAT) status = DftiSetValue(hand, DFTI_INPUT_STRIDES, & <real_strides>) status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, & <complex_strides>) status = DftiCommitDescriptor(hand) ! Compute an FFT status = DftiComputeForward(hand, X_inout) </pre>
Real-to-complex, out-of-place, forward	Real numbers	Numbers in the CCE format	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision> &, DFTI_REAL, <dimension>, <sizes>) status = DftiSetValue(hand, & DFTI_CONJUGATE_EVEN_STORAGE, & DFTI_COMPLEX_COMPLEX) status = DftiSetValue(hand, DFTI_PACKED_FORMAT, & DFTI_CCE_FORMAT) status = DftiSetValue(hand, DFTI_PLACEMENT, & DFTI_NOT_INPLACE) status = DftiSetValue(hand, DFTI_INPUT_STRIDES, & <real_strides>) status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, & <complex_strides>) status = DftiCommitDescriptor(hand) ! Compute an FFT status = DftiComputeForward(hand, X_in, Y_out) </pre>
Complex-to-real, in-place, backward	Numbers in the CCE format	Real numbers	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_REAL, <dimension>, <sizes>) status = DftiSetValue(hand, & DFTI_CONJUGATE_EVEN_STORAGE, & DFTI_COMPLEX_COMPLEX) status = DftiSetValue(hand, DFTI_PACKED_FORMAT, & DFTI_CCE_FORMAT) status = DftiSetValue(hand, DFTI_INPUT_STRIDES, & <complex_strides>) status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, & <real_strides>) status = DftiCommitDescriptor(hand) ! Compute an FFT status = DftiComputeBackward(hand, X_inout) </pre>
Complex-to-real, out-of-place, backward	Numbers in the CCE format	Real numbers	<pre> ! Configure a Descriptor status = DftiCreateDescriptor(hand, <precision>, & DFTI_REAL, <dimension>, <sizes>) </pre>

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
			<pre> status = DftiSetValue(hand, & DFTI_CONJUGATE_EVEN_STORAGE, & DFTI_COMPLEX_COMPLEX) status = DftiSetValue(hand, DFTI_PLACEMENT, & DFTI_NOT_INPLACE) status = DftiSetValue(hand, DFTI_PACKED_FORMAT, & DFTI_CCE_FORMAT) status = DftiSetValue(hand, DFTI_INPUT_STRIDES, & <complex_strides>) status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, & <real_strides>) status = DftiCommitDescriptor(hand) ! Compute an FFT status = DftiComputeBackward(hand, X_in, Y_out) </pre>

You can find Fortran programs that illustrate configuring and computing FFTs in the examples/dftf/ subdirectory of your Intel MKL directory.

Configuring and Computing an FFT in C/C++

The table below summarizes information on configuring and computing an FFT in C/C++ for all kinds of transforms and possible combinations of input and output domains.

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
Complex-to-complex, in-place, forward or backward	Interleaved complex numbers	Interleaved complex numbers	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_COMPLEX, <dimension>, <sizes>); status = DftiCommitDescriptor(hand); /* Compute an FFT */ /* forward FFT */ status = DftiComputeForward(hand, X_inout); /* or backward FFT */ status = DftiComputeBackward(hand, X_inout); </pre>
Complex-to-complex, out-of-place, forward or backward	Interleaved complex numbers	Interleaved complex numbers	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_COMPLEX, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_PLACEMENT, DFTI_NOT_INPLACE); status = DftiCommitDescriptor(hand); /* Compute an FFT */ /* forward FFT */ status = DftiComputeForward(hand, X_in, Y_out); /* or backward FFT */ status = DftiComputeBackward(hand, X_in, Y_out); </pre>
Complex-to-complex, in-place, forward or backward	Split-complex numbers	Split-complex numbers	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_COMPLEX, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_COMPLEX_STORAGE, DFTI_REAL_REAL); status = DftiCommitDescriptor(hand); </pre>

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
Complex-to-complex, out-of-place, forward or backward	Split-complex numbers	Split-complex numbers	<pre> /* Compute an FFT */ /* forward FFT */ status = DftiComputeForward(hand, Xre_inout, Xim_inout); /* or backward FFT */ status = DftiComputeBackward(hand, Xre_inout, Xim_inout); /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_COMPLEX, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_COMPLEX_STORAGE, DFTI_REAL_REAL); status = DftiSetValue(hand, DFTI_PLACEMENT, DFTI_NOT_INPLACE); status = DftiCommitDescriptor(hand); /* Compute an FFT */ /* forward FFT */ status = DftiComputeForward(hand, Xre_in, Xim_in, Yre_out, Yim_out); /* or backward FFT */ status = DftiComputeBackward(hand, Xre_in, Xim_in, Yre_out, Yim_out); </pre>
Real-to-complex, in-place, forward	Real numbers	Numbers in the CCE format	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_REAL, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX); status = DftiSetValue(hand, DFTI_PACKED_FORMAT, DFTI_CCE_FORMAT); status = DftiSetValue(hand, DFTI_INPUT_STRIDES, <real_strides>); status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, <complex_strides>); status = DftiCommitDescriptor(hand); /* Compute an FFT */ status = DftiComputeForward(hand, X_inout); </pre>
Real-to-complex, out-of-place, forward	Real numbers	Numbers in the CCE format	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_REAL, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX); status = DftiSetValue(hand, DFTI_PACKED_FORMAT, DFTI_CCE_FORMAT); status = DftiSetValue(hand, DFTI_PLACEMENT, DFTI_NOT_INPLACE); status = DftiSetValue(hand, DFTI_INPUT_STRIDES, <real_strides>); status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, </pre>

FFT to Compute	Input Data	Output Data	Required FFT Function Calls
Complex-to-real, in-place, backward	Numbers in the CCE format	Real numbers	<pre> <complex_strides>; status = DftiCommitDescriptor(hand); /* Compute an FFT */ status = DftiComputeForward(hand, X_in, Y_out); </pre>
Complex-to-real, out-of-place, backward	Numbers in the CCE format	Real numbers	<pre> /* Configure a Descriptor */ status = DftiCreateDescriptor(&hand, <precision>, DFTI_REAL, <dimension>, <sizes>); status = DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX); status = DftiSetValue(hand, DFTI_PACKED_FORMAT, DFTI_CCE_FORMAT); status = DftiSetValue(hand, DFTI_INPUT_STRIDES, <complex_strides>); status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, <real_strides>); status = DftiCommitDescriptor(hand); /* Compute an FFT */ status = DftiComputeBackward(hand, X_inout); </pre>

You can find C programs that illustrate configuring and computing FFTs in the examples/dftc/ subdirectory of your Intel MKL directory.

Status Checking Functions

All of the descriptor manipulation, FFT computation, and descriptor configuration functions return an integer value denoting the status of the operation. The functions in this category check that status. The first function is a logical function that checks whether the status reflects an error of a predefined class, and the second is an error message function that returns a character string.

DftiErrorClass

Checks whether the status reflects an error of a predefined class.

Syntax

Fortran:

```
predicate = DftiErrorClass( status, error_class )
```

C:

```
predicate = DftiErrorClass(status, error_class);
```

Include Files

- Fortran 90: mkl_dfti.f90
- C: mkl.h

Input Parameters

Name	Type	Description
status	<small>FORTRAN:</small> INTEGER <small>C:</small> MKL_LONG	Completion status of an FFT function.
error_class	<small>FORTRAN:</small> INTEGER <small>C:</small> MKL_LONG	Predefined error class.

Output Parameters

Name	Type	Description
<i>predicate</i>	<small>FORTRAN:</small> LOGICAL <small>C:</small> MKL_LONG	Result of checking.

Description

The FFT interface in Intel MKL provides a set of predefined error classes listed in [Table "Predefined Error Classes"](#). They are named constants and have the type INTEGER in Fortran and MKL_LONG in C.

Predefined Error Classes

Named Constants	Comments
DFTI_NO_ERROR	No error. The zero status belongs to this class.
DFTI_MEMORY_ERROR	Usually associated with memory allocation
DFTI_INVALID_CONFIGURATION	Invalid settings of one or more configuration parameters
DFTI_INCONSISTENT_CONFIGURATION	Inconsistent configuration or input parameters
DFTI_NUMBER_OF_THREADS_ERROR	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function)
DFTI_MULTITHREADED_ERROR	Usually associated with a value that OMP routines return in case of errors

Named Constants	Comments
DFTI_BAD_DESCRIPTOR	Descriptor is unusable for computation
DFTI_UNIMPLEMENTED	Unimplemented legitimate settings; implementation dependent
DFTI_MKL_INTERNAL_ERROR	Internal library error
DFTI_1D_LENGTH_EXCEEDS_INT32	Length of one of dimensions exceeds $2^{32} - 1$ (4 bytes).

The `DftiErrorClass` function returns a non-zero value in C or the value of `.TRUE.` in Fortran if the status belongs to the predefined error class. To check whether a function call was successful, call `DftiErrorClass` with a specific error class. However, the zero value of the status belongs to the `DFTI_NO_ERROR` class and thus the zero status indicates successful completion of an operation. See [Example "Using Status Checking Functions"](#) for an illustration of correct use of the status checking functions.

NOTE

It is incorrect to directly compare a status with a predefined class.

Interface

```
//Fortran interface
INTERFACE DftiErrorClass
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_8( Status, Error_Class )
LOGICAL some_actual_function_8
INTEGER, INTENT(IN) :: Status, Error_Class
END FUNCTION some_actual_function_8
END INTERFACE DftiErrorClass
```

Prototype

```
/* C prototype */
MKL_LONG DftiErrorClass( MKL_LONG , MKL_LONG );
```

DftiErrorMessage

Generates an error message.

Syntax

Fortran:

```
error_message = DftiErrorMessage( status )
```

C:

```
error_message = DftiErrorMessage(status);
```

Include Files

- Fortran 90: `mkl_dfti.f90`

- C: mkl.h

Input Parameters

Name	Type	Description
status	FORTRAN : INTEGER C : MKL_LONG	Completion status of a function.

Output Parameters

Name	Type	Description
error_message	FORTRAN : CHARACTER (LEN=DFTI_MAX_MESSAGE_LENGTH) C : Array of char	The character string with the error message.

Description

The error message function generates an error message character string. In Fortran, use a character string of length DFTI_MAX_MESSAGE_LENGTH as a target for the error message. In C, the function returns a pointer to a constant character string, that is, a character array with terminating '\0' character, and you do not need to free this pointer.

[Example "Using Status Checking Function"](#) shows how this function can be used.

Interface

```
//Fortran interface
INTERFACE DftiErrorMessage
//Note that the body provided here is to illustrate the different
//argument list and types of dummy arguments. The interface
//does not guarantee what the actual function names are.
//Users can only rely on the function name following the
//keyword INTERFACE
FUNCTION some_actual_function_9( Status )
CHARACTER(LEN=DFTI_MAX_MESSAGE_LENGTH) some_actual_function_9( Status )
INTEGER, INTENT(IN) :: Status
END FUNCTION some_actual_function_9
END INTERFACE DftiErrorMessage
```

Prototype

```
/* C prototype */
char *DftiErrorMessage( MKL_LONG );
```

Cluster FFT Functions

This section describes the cluster Fast Fourier Transform (FFT) functions implemented in Intel® MKL.

NOTE

These functions are available only for the Linux* and Windows* operating systems.

The cluster FFT function library was designed to perform fast Fourier transforms on a cluster, that is, a group of computers interconnected via a network. Each computer (node) in the cluster has its own memory and processor(s). Data interchanges between the nodes are provided by the network.

One or more processes may be running in parallel on each cluster node. To organize communication between different processes, the cluster FFT function library uses the Message Passing Interface (MPI). To avoid dependence on a specific MPI implementation (for example, MPICH, Intel® MPI, and others), the library works with MPI via a message-passing library for linear algebra called BLACS.

Cluster FFT functions of Intel MKL provide one-dimensional, two-dimensional, and multi-dimensional (up to the order of 7) functions and both Fortran and C interfaces for all transform functions.

To develop applications using the cluster FFT functions, you should have basic skills in MPI programming.

The interfaces for the Intel MKL cluster FFT functions are similar to the corresponding interfaces for the conventional Intel MKL [FFT functions](#), described earlier in this chapter. Refer there for details not explained in this section.

Table "Cluster FFT Functions in Intel MKL" lists cluster FFT functions implemented in Intel MKL:

Cluster FFT Functions in Intel MKL

Function Name	Operation
Descriptor Manipulation Functions	
DftiCreateDescriptorDM	Allocates memory for the descriptor data structure and preliminarily initializes it.
DftiCommitDescriptorDM	Performs all initialization for the actual FFT computation.
DftiFreeDescriptorDM	Frees memory allocated for a descriptor.
FFT Computation Functions	
DftiComputeForwardDM	Computes the forward FFT.
DftiComputeBackwardDM	Computes the backward FFT.
Descriptor Configuration Functions	
DftiSetValueDM	Sets one particular configuration parameter with the specified configuration value.
DftiGetValueDM	Gets the value of one particular configuration parameter.

Computing Cluster FFT

The cluster FFT functions described later in this section are provided with Fortran and C interfaces. Fortran stands for Fortran 95.

Cluster FFT computation is performed by [DftiComputeForwardDM](#) and [DftiComputeBackwardDM](#) functions, called in a program using MPI, which will be referred to as MPI program. After an MPI program starts, a number of processes are created. MPI identifies each process by its rank. The processes are independent of one another and communicate via MPI. A function called in an MPI program is invoked in all the processes. Each process manipulates data according to its rank. Input or output data for a cluster FFT transform is a sequence of real or complex values. A cluster FFT computation function operates on the local part of the input data, that is, some part of the data to be operated in a particular process, as well as generates local part of the output data. While each process performs its part of computations, running in parallel and communicating through MPI, the processes perform the entire FFT computation. FFT computations using the Intel MKL cluster FFT functions are typically effected by a number of steps listed below:

1. Initiate MPI by calling `MPI_Init` in C/C++ or `MPI_INIT` in Fortran (the function must be called prior to calling any FFT function and any MPI function).
2. Allocate memory for the descriptor and create it by calling [DftiCreateDescriptorDM](#).
3. Specify one of several values of configuration parameters by one or more calls to [DftiSetValueDM](#).

4. Obtain values of configuration parameters needed to create local data arrays; the values are retrieved by calling [DftiGetValueDM](#).
5. Initialize the descriptor for the FFT computation by calling [DftiCommitDescriptorDM](#).
6. Create arrays for local parts of input and output data and fill the local part of input data with values. (For more information, see [Distributing Data among Processes](#).)
7. Compute the transform by calling [DftiComputeForwardDM](#) or [DftiComputeBackwardDM](#).
8. Gather local output data into the global array using MPI functions. (This step is optional because you may need to immediately employ the data differently.)
9. Release memory allocated for the descriptor by calling [DftiFreeDescriptorDM](#).
10. Finalize communication through MPI by calling `MPI_Finalize` in C/C++ or `MPI_FINALIZE` in Fortran (the function must be called after the last call to a cluster FFT function and the last call to an MPI function).

Several code examples in the "[Examples for Cluster FFT Functions](#)" section in the Code Examples appendix illustrate cluster FFT computations.

Distributing Data among Processes

The Intel MKL cluster FFT functions store all input and output multi-dimensional arrays (matrices) in one-dimensional arrays (vectors). The arrays are stored in the row-major order in C/C++ and in the column-major order in Fortran. For example, a two-dimensional matrix A of size (m,n) is stored in a vector B of size $m*n$ so that

$B[i*n+j]=A[i][j]$ in C/C++ ($i=0, \dots, m-1, j=0, \dots, n-1$)

and

$B((j-1)*m+i)=A(i,j)$ in Fortran ($i=1, \dots, m, j=1, \dots, n$) .

NOTE

Order of FFT dimensions is the same as the order of array dimensions in the programming language. For example, a 3-dimensional FFT with `Lengths=(m,n,l)` can be computed over an array `Ar[m][n][l]` in C/C++ or `AR(m,n,l)` in Fortran.

All MPI processes involved in cluster FFT computation operate their own portions of data. These local arrays make up the virtual global array that the fast Fourier transform is applied to. It is your responsibility to properly allocate local arrays (if needed), fill them with initial data and gather resulting data into an actual global array or process the resulting data differently. To be able do this, see sections below on how the virtual global array is composed of the local ones.

Multi-dimensional transforms

If the dimension of transform is greater than one, the cluster FFT function library splits data in the dimension whose index changes most slowly, so that the parts contain all elements with several consecutive values of this index. It is the first dimension in C and the last dimension in Fortran. If the global array is two-dimensional, in C, it gives each process several consecutive rows. The term "rows" will be used regardless of the array dimension and programming language. Local arrays are placed in memory allocated for the virtual global array consecutively, in the order determined by process ranks. For example, in case of two processes, during the computation of a three-dimensional transform whose matrix has size $(11,15,12)$, the processes may store local arrays of sizes $(6,15,12)$ and $(5,15,12)$, respectively.

If p is the number of MPI processes and the matrix of a transform to be computed has size (m,n,l) , in C, each MPI process works with local data array of size (m_q, n, l) , where $\sum m_q=m$, $q=0, \dots, p-1$. Local input arrays must contain appropriate parts of the actual global input array, and then local output arrays will contain appropriate parts of the actual global output array. You can figure out which particular rows of the global array the local array must contain from the following configuration parameters of the cluster FFT interface: `CDFT_LOCAL_NX`, `CDFT_LOCAL_START_X`, and `CDFT_LOCAL_SIZE`. To retrieve values of the parameters, use the [DftiGetValueDM](#) function:

- `CDFT_LOCAL_NX` specifies how many rows of the global array the current process receives.
- `CDFT_LOCAL_START_X` specifies which row of the global input or output array corresponds to the first row of the local input or output array. If `A` is a global array and `L` is the appropriate local array, then

`L[i][j][k]=A[i+cdft_local_start_x][j][k]`, where $i=0, \dots, m_q-1, j=0, \dots, n-1, k=0, \dots, l-1$, for C/C++

and

`L(i,j,k)=A(i,j,k+cdft_local_start_x-1)`, where $i=1, \dots, m, j=1, \dots, n, k=1, \dots, l_q$, for Fortran.

[Example "2D Out-of-place Cluster FFT Computation" in the Code Examples appendix](#) shows how the data is distributed among processes for a two-dimensional cluster FFT computation.

One-dimensional transforms

In this case, input and output data are distributed among processes differently and even the numbers of elements stored in a particular process before and after the transform may be different. Each local array stores a segment of consecutive elements of the appropriate global array. Such segment is determined by the number of elements and a shift with respect to the first array element. So, to specify segments of the global input and output arrays that a particular process receives, *four* configuration parameters are needed: `CDFT_LOCAL_NX`, `CDFT_LOCAL_START_X`, `CDFT_LOCAL_OUT_NX`, and `CDFT_LOCAL_OUT_START_X`. Use the `DftiGetValueDM` function to retrieve their values. The meaning of the four configuration parameters depends upon the type of the transform, as shown in [Table "Data Distribution Configuration Parameters for 1D Transforms"](#):

Data Distribution Configuration Parameters for 1D Transforms

Meaning of the Parameter	Forward Transform	Backward Transform
Number of elements in input array	<code>CDFT_LOCAL_NX</code>	<code>CDFT_LOCAL_OUT_NX</code>
Elements shift in input array	<code>CDFT_LOCAL_START_X</code>	<code>CDFT_LOCAL_OUT_START_X</code>
Number of elements in output array	<code>CDFT_LOCAL_OUT_NX</code>	<code>CDFT_LOCAL_NX</code>
Elements shift in output array	<code>CDFT_LOCAL_OUT_START_X</code>	<code>CDFT_LOCAL_START_X</code>

Memory size for local data

The memory size needed for local arrays cannot be just calculated from `CDFT_LOCAL_NX` (`CDFT_LOCAL_OUT_NX`), because the cluster FFT functions sometimes require allocating a little bit more memory for local data than just the size of the appropriate sub-array. The configuration parameter `CDFT_LOCAL_SIZE` specifies the size of the local input and output array in data elements. Each local input and output arrays must have size not less than `CDFT_LOCAL_SIZE*size_of_element`. Note that in the current implementation of the cluster FFT interface, data elements can be real or complex values, each complex value consisting of the real and imaginary parts. If you employ a user-defined workspace for in-place transforms (for more information, refer to [Table "Settable configuration Parameters"](#)), it must have the same size as the local arrays. [Example "1D In-place Cluster FFT Computations" in the Code Examples appendix](#) illustrates how the cluster FFT functions distribute data among processes in case of a one-dimensional FFT computation performed with a user-defined workspace.

Available Auxiliary Functions

If a global input array is located on one MPI process and you want to obtain its local parts or you want to gather the global output array on one MPI process, you can use functions `MKL_CDFT_ScatterData` and `MKL_CDFT_GatherData` to distribute or gather data among processes, respectively. These functions are

defined in a file that is delivered with Intel MKL and located in the following subdirectory of the Intel MKL installation directory: `examples/cdftc/source/cdft_example_support.c` for C/C++ and `examples/cdftf/source/cdft_example_support.f90` for Fortran.

Restriction on Lengths of Transforms

The algorithm that the Intel MKL cluster FFT functions use to distribute data among processes imposes a restriction on lengths of transforms with respect to the number of MPI processes used for the FFT computation:

- For a multi-dimensional transform, lengths of the first two dimensions in C/C++ or of the last two dimensions in Fortran must be not less than the number of MPI processes.
- Length of a one-dimensional transform must be the product of two integers each of which is not less than the number of MPI processes.

Non-compliance with the restriction causes an error `CDFT_SPREAD_ERROR` (refer to [Error Codes](#) for details).

To achieve the compliance, you can change the transform lengths and/or the number of MPI processes, which is specified at start of an MPI program. MPI-2 enables changing the number of processes during execution of an MPI program.

Cluster FFT Interface

To use the cluster FFT functions, you need to access the module `MKL_CDFT` through the "use" statement in Fortran; or access the header file `mkl_cdft.h` through "include" in C/C++.

The Fortran interface provides a derived type `DFTI_DESCRIPTOR_DM`; a number of named constants representing various names of configuration parameters and their possible values; and a number of overloaded functions through the generic functionality of Fortran 95.

The C interface provides a structure type `DFTI_DESCRIPTOR_DM_HANDLE` and a number of functions, some of which accept a different number of input arguments.

To provide communication between parallel processes through MPI, the following include statement must be present in your code:

- Fortran:

```
INCLUDE "mpif.h"
(for some MPI versions, "mpif90.h" header may be used instead).
```

- C/C++:

```
#include "mpi.h"
```

There are three main categories of the cluster FFT functions in Intel MKL:

1. **Descriptor Manipulation.** There are three functions in this category. The `DftiCreateDescriptorDM` function creates an FFT descriptor whose storage is allocated dynamically. The `DftiCommitDescriptorDM` function "commits" the descriptor to all its settings. The `DftiFreeDescriptorDM` function frees up the memory allocated for the descriptor.
2. **FFT Computation.** There are two functions in this category. The `DftiComputeForwardDM` function performs the forward FFT computation, and the `DftiComputeBackwardDM` function performs the backward FFT computation.
3. **Descriptor Configuration.** There are two functions in this category. The `DftiSetValueDM` function sets one specific configuration value to one of the many configuration parameters. The `DftiGetValueDM` function gets the current value of any of these configuration parameters, all of which are readable. These parameters, though many, are handled one at a time.

Descriptor Manipulation Functions

There are three functions in this category: create a descriptor, commit a descriptor, and free a descriptor.

DftiCreateDescriptorDM

Allocates memory for the descriptor data structure and preliminarily initializes it.

Syntax

Fortran:

```
Status = DftiCreateDescriptorDM(comm, handle, v1, v2, dim, size)
Status = DftiCreateDescriptorDM(comm, handle, v1, v2, dim, sizes)
```

C:

```
status = DftiCreateDescriptorDM(comm, &handle, v1, v2, dim, size);
status = DftiCreateDescriptorDM(comm, &handle, v1, v2, dim, sizes);
```

Include Files

- Fortran 90: mkl_cdft.f90
- C: mkl_cdft.h

Input Parameters

<i>comm</i>	MPI communicator, e.g. MPI_COMM_WORLD.
<i>v1</i>	Precision of the transform.
<i>v2</i>	Type of the forward domain. Must be DFTI_COMPLEX for complex-to-complex transforms or DFTI_REAL for real-to-complex transforms.
<i>dim</i>	Dimension of the transform.
<i>size</i>	Length of the transform in a one-dimensional case.
<i>sizes</i>	Lengths of the transform in a multi-dimensional case.

Output Parameters

<i>handle</i>	Pointer to the descriptor handle of transform. If the function completes successfully, the pointer to the created handle is stored in the variable.
---------------	---

Description

This function allocates memory in a particular MPI process for the descriptor data structure and instantiates it with default configuration settings with respect to the precision, domain, dimension, and length of the desired transform. The domain is understood to be the domain of the forward transform. The result is a pointer to the created descriptor. This function is slightly different from the "initialization" function [DftiCommitDescriptorDM](#) in a more traditional software packages or libraries used for computing the FFT. This function does not perform any significant computation work, such as twiddle factors computation, because the default configuration settings can still be changed using the function [DftiSetValueDM](#).

The value of the parameter *v1* is specified through named constants DFTI_SINGLE and DFTI_DOUBLE. It corresponds to precision of input data, output data, and computation. A setting of DFTI_SINGLE indicates single-precision floating-point data type and a setting of DFTI_DOUBLE indicates double-precision floating-point data type.

The parameter *dim* is a simple positive integer indicating the dimension of the transform.

In C/C++, for one-dimensional transforms, length is a single integer value of the parameter *size* having type MKL_LONG; for multi-dimensional transforms, length is supplied with the parameter *sizes*, which is an array of integers having type MKL_LONG. In Fortran, length is an integer or an array of integers.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. In this case, the pointer to the created descriptor handle is stored in `handle`. If the function fails, it returns a value of another error class constant

Interface

```
! Fortran interface
INTERFACE DftiCreateDescriptorDM
    INTEGER(4) FUNCTION DftiCreateDescriptorDMn(C,H,P1,P2,D,L)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: H
        INTEGER(4) C,P1,P2,D,L(*)
    END FUNCTION
    INTEGER(4) FUNCTION DftiCreateDescriptorDM1(C,H,P1,P2,D,L)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: H
        INTEGER(4) C,P1,P2,D,L
    END FUNCTION
END INTERFACE
```

Prototype

```
/* C/C++ prototype */
MKL_LONG DftiCreateDescriptorDM(MPI_Comm,DFTI_DESCRIPTOR_DM_HANDLE*,
    enum DFTI_CONFIG_VALUE,enum DFTI_CONFIG_VALUE,MKL_LONG,...);
```

DftiCommitDescriptorDM

Performs all initialization for the actual FFT computation.

Syntax

Fortran:

```
Status = DftiCommitDescriptorDM(handle)
```

C:

```
status = DftiCommitDescriptorDM(handle);
```

Include Files

- Fortran 90: `mkl_cdft.f90`
- C: `mkl_cdft.h`

Input Parameters

<code>handle</code>	The descriptor handle. Must be valid, that is, created in a call to DftiCreateDescriptorDM .
---------------------	--

Description

The cluster FFT interface requires a function that completes initialization of a previously created descriptor before the descriptor can be used for FFT computations in a particular MPI process. The `DftiCommitDescriptorDM` function performs all initialization that facilitates the actual FFT computation. For the current implementation, it may involve exploring many different factorizations of the input length to search for highly efficient computation method.

Any changes of configuration parameters of a committed descriptor via the set value function (see [Descriptor Configuration Functions](#)) requires a re-committal of the descriptor before a computation function can be invoked. Typically, this committal function is called right before a computation function call (see [FFT Computation Functions](#)).

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface

```
! Fortran interface
INTERFACE DftiCommitDescriptorDM
    INTEGER(4) FUNCTION DftiCommitDescriptorDM(handle);
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: handle
    END FUNCTION
END INTERFACE
```

Prototype

```
/* C/C++ prototype */
MKL_LONG DftiCommitDescriptorDM(DFTI_DESCRIPTOR_DM_HANDLE handle);
```

DftiFreeDescriptorDM

Frees memory allocated for a descriptor.

Syntax

Fortran:

```
Status = DftiFreeDescriptorDM(handle)
```

C:

```
status = DftiFreeDescriptorDM(&handle);
```

Include Files

- Fortran 90: `mkl_cdft.f90`
- C: `mkl_cdft.h`

Input Parameters

<i>handle</i>	The descriptor handle. Must be valid, that is, created in a call to DftiCreateDescriptorDM .
---------------	--

Output Parameters

<i>handle</i>	The descriptor handle. Memory allocated for the handle is released on output.
---------------	---

Description

This function frees up all memory allocated for a descriptor in a particular MPI process. Call the `DftiFreeDescriptorDM` function to delete the descriptor handle. Upon successful completion of `DftiFreeDescriptorDM` the descriptor handle is no longer valid.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface

```
! Fortran interface
INTERFACE DftiFreeDescriptorDM
  INTEGER(4) FUNCTION DftiFreeDescriptorDM(handle)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: handle
  END FUNCTION
END INTERFACE
```

Prototype

```
/* C/C++ prototype */
MKL_LONG DftiFreeDescriptorDM(DFTI_DESCRIPTOR_DM_HANDLE *handle);
```

FFT Computation Functions

There are two functions in this category: compute the forward transform and compute the backward transform.

DftiComputeForwardDM

Computes the forward FFT.

Syntax

Fortran:

```
Status = DftiComputeForwardDM(handle, in_X, out_X)
Status = DftiComputeForwardDM(handle, in_out_X)
```

C:

```
status = DftiComputeForwardDM(handle, in_X, out_X);
status = DftiComputeForwardDM(handle, in_out_X);
```

Include Files

- Fortran 90: `mkl_cdft.f90`
- C: `mkl_cdft.h`

Input Parameters

<code>handle</code>	The descriptor handle.
<code>in_X, in_out_X</code>	Local part of input data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate and initialize the array.

Output Parameters

out_X, *in_out_X*

Local part of output data. Array of complex values. Refer to the [Distributing Data among Processes](#) section on how to allocate the array.

Description

The `DftiComputeForwardDM` function computes the forward FFT. Forward FFT is the transform using the factor $e^{-i2\pi/n}$.

Before you call the function, the valid descriptor, created by `DftiCreateDescriptorDM`, must be configured and committed using the `DftiCommitDescriptorDM` function.

The computation is carried out by calling the `DftiComputeForward` function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of `DftiComputeForward`.

Local part of input data, as well as local part of the output data, is an appropriate sequence of complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See the [Distributing Data Among Processes](#) section for details.

Refer to the [Configuration Settings](#) section for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter `DFTI_PRECISION` determines the precision of input data, output data, and transform: a setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The configuration parameter `DFTI_PLACEMENT` informs the function whether the computation should be in-place. If the value of this parameter is `DFTI_INPLACE` (default), you must call the function with two parameters, otherwise you must supply three parameters. If `DFTI_PLACEMENT = DFTI_INPLACE` and three parameters are supplied, then the third parameter is ignored.

CAUTION

Even in case of an out-of-place transform, local array of input data *in_X* may be changed. To save data, make its copy before calling `DftiComputeForwardDM`.

In case of an in-place transform, `DftiComputeForwardDM` dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.

NOTE

You can specify your own workspace of the same size through the configuration parameter `CDFI_WORKSPACE` to avoid redundant memory allocation.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface

```

! Fortran interface
INTERFACE DftiComputeForwardDM
  INTEGER(4) FUNCTION DftiComputeForwardDM(h, in_X, out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(8), DIMENSION(*) :: in_X, out_X
  END FUNCTION DftiComputeForwardDM
END INTERFACE

```

```

END FUNCTION DftiComputeForwardDM
INTEGER(4) FUNCTION DftiComputeForwardDMi(h, in_out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(8), DIMENSION(*) :: in_out_X
END FUNCTION DftiComputeForwardDMi
INTEGER(4) FUNCTION DftiComputeForwardDMs(h, in_X, out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(4), DIMENSION(*) :: in_X, out_X
END FUNCTION DftiComputeForwardDMs
INTEGER(4) FUNCTION DftiComputeForwardDMIs(h, in_out_X)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(4), DIMENSION(*) :: in_out_X
END FUNCTION DftiComputeForwardDMIs
END INTERFACE

```

Prototype

```

/* C/C++ prototype */
MKL_LONG DftiComputeForwardDM(DFTI_DESCRIPTOR_DM_HANDLE handle, void *in_X,...);

```

DftiComputeBackwardDM

Computes the backward FFT.

Syntax

Fortran:

```

Status = DftiComputeBackwardDM(handle, in_X, out_X)
Status = DftiComputeBackwardDM(handle, in_out_X)

```

C:

```

status = DftiComputeBackwardDM(handle, in_X, out_X);
status = DftiComputeBackwardDM(handle, in_out_X);

```

Include Files

- Fortran 90: mkl_cdft.f90
- C: mkl_cdft.h

Input Parameters

<i>handle</i>	The descriptor handle.
<i>in_X, in_out_X</i>	Local part of input data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate and initialize the array.

Output Parameters

<i>out_X, in_out_X</i>	Local part of output data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate the array.
------------------------	--

Description

The `DftiComputeBackwardDM` function computes the backward FFT. Backward FFT is the transform using the factor $e^{i2\pi/n}$.

Before you call the function, the valid descriptor, created by `DftiCreateDescriptorDM`, must be configured and committed using the `DftiCommitDescriptorDM` function.

The computation is carried out by calling the `DftiComputeBackward` function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of `DftiComputeBackward`.

Local part of input data, as well as local part of the output data, is an appropriate sequence of complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See the [Distributing Data among Processes](#) section for details.

Refer to the [Configuration Settings](#) section for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter `DFTI_PRECISION` determines the precision of input data, output data, and transform: a setting of `DFTI_SINGLE` indicates single-precision floating-point data type and a setting of `DFTI_DOUBLE` indicates double-precision floating-point data type.

The configuration parameter `DFTI_PLACEMENT` informs the function whether the computation should be in-place. If the value of this parameter is `DFTI_INPLACE` (default), you must call the function with two parameters, otherwise you must supply three parameters. If `DFTI_PLACEMENT = DFTI_INPLACE` and three parameters are supplied, then the third parameter is ignored.

CAUTION

Even in case of an out-of-place transform, local array of input data `in_X` may be changed. To save data, make its copy before calling `DftiComputeBackwardDM`.

In case of an in-place transform, `DftiComputeBackwardDM` dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.

NOTE

You can specify your own workspace of the same size through the configuration parameter `CDFT_WORKSPACE` to avoid redundant memory allocation.

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface

```

! Fortran interface
INTERFACE DftiComputeBackwardDM
    INTEGER(4) FUNCTION DftiComputeBackwardDM(h, in_X, out_X)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        COMPLEX(8), DIMENSION(*) :: in_X, out_X
    END FUNCTION DftiComputeBackwardDM
    INTEGER(4) FUNCTION DftiComputeBackwardDMi(h, in_out_X)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        COMPLEX(8), DIMENSION(*) :: in_out_X
    END FUNCTION DftiComputeBackwardDMi
    INTEGER(4) FUNCTION DftiComputeBackwardDMs(h, in_X, out_X)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        COMPLEX(4), DIMENSION(*) :: in_X, out_X
    END FUNCTION DftiComputeBackwardDMs
    INTEGER(4) FUNCTION DftiComputeBackwardDMis(h, in_out_X)

```

```

    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    COMPLEX(4), DIMENSION(*) :: in_out_X
  END FUNCTION DftiComputeBackwardDMIs
END INTERFACE

```

Prototype

```

/* C/C++ prototype */
MKL_LONG DftiComputeBackwardDM(DFTI_DESCRIPTOR_DM_HANDLE handle, void *in_X,...);

```

Descriptor Configuration Functions

There are two functions in this category: the value setting function [DftiSetValueDM](#) sets one particular configuration parameter to an appropriate value, the value getting function [DftiGetValueDM](#) reads the value of one particular configuration parameter.

Some configuration parameters used by cluster FFT functions originate from the conventional FFT interface (see [Configuration Settings](#) subsection in the "FFT Functions" section for details).

Other configuration parameters are specific to the cluster FFT. Integer values of these parameters have type MKL_LONG in C/C++ and INTEGER(4) in Fortran. The exact type of the configuration parameters being floating-point scalars is float or double in C/C++ and REAL(4) or REAL(8) in Fortran. The configuration parameters whose values are named constants have the enum type in C/C++ and INTEGER type in Fortran. They are defined in the mkl_cdft.h header file in C/C++ and MKL_CDFT module in Fortran.

Names of the configuration parameters specific to the cluster FFT interface have prefix CDFT.

DftiSetValueDM

Sets one particular configuration parameter with the specified configuration value.

Syntax

Fortran:

```
Status = DftiSetValueDM (handle, param, value)
```

C:

```
status = DftiSetValueDM (handle, param, value);
```

Include Files

- Fortran 90: mkl_cdft.f90
- C: mkl_cdft.h

Input Parameters

<i>handle</i>	The descriptor handle. Must be valid, that is, created in a call to DftiCreateDescriptorDM .
<i>param</i>	Name of a parameter to be set up in the descriptor handle. See Table "Settable Configuration Parameters" for the list of available parameters.
<i>value</i>	Value of the parameter.

Description

This function sets one particular configuration parameter with the specified configuration value. The configuration parameter is one of the named constants listed in the table below, and the configuration value must have the corresponding type. See [Configuration Settings](#) for details of the meaning of each setting and for possible values of the parameters whose values are named constants.

Settable Configuration Parameters

Parameter Name	Data Type	Description	Default Value
DFTI_FORWARD_SCALE	Floating-point scalar	Scale factor of forward transform.	1.0
DFTI_BACKWARD_SCALE	Floating-point scalar	Scale factor of backward transform.	1.0
DFTI_PLACEMENT	Named constant	Placement of the computation result.	DFTI_INPLACE
DFTI_ORDERING	Named constant	Scrambling of data order.	DFTI_ORDERED
CDFT_WORKSPACE	Array of an appropriate type	Auxiliary buffer, a user-defined workspace. Enables saving memory during in-place computations.	NULL (allocate workspace dynamically).
DFTI_PACKED_FORMAT	Named constant	Packed format, real data.	<ul style="list-style-type: none"> • DFTI_PERM_FORMAT — default and the only available value for one-dimensional transforms • DFTI_CCE_FORMAT — default and the only available value for multi-dimensional transforms
DFTI_TRANSPOSE	Named constant	This parameter determines how the output data is located for multi-dimensional transforms. If the parameter value is DFTI_NONE, the data is located in a usual manner described in this manual. If the value is DFTI_ALLOW, the last (first) global transposition is not performed for a forward (backward) transform.	DFTI_NONE

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface

```
! Fortran interface
INTERFACE DftiSetValueDM
```

```

INTEGER(4) FUNCTION DftiSetValueDM(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p, v
END FUNCTION
INTEGER(4) FUNCTION DftiSetValueDMd(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p
    REAL(8) :: v
END FUNCTION
INTEGER(4) FUNCTION DftiSetValueDMs(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p
    REAL(4) :: v
END FUNCTION
INTEGER(4) FUNCTION DftiSetValueDMsw(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p
    COMPLEX(4) :: v(*)
END FUNCTION
INTEGER(4) FUNCTION DftiSetValueDMdw(h, p, v)
    TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
    INTEGER(4) :: p
    COMPLEX(8) :: v(*)
END FUNCTION
END INTERFACE

```

Prototype

```

/* C/C++ prototype */
MKL_LONG DftiSetValueDM(DFTI_DESCRIPTOR_DM_HANDLE handle, int param,...);

```

DftiGetValueDM

Gets the value of one particular configuration parameter.

Syntax

Fortran:

```
Status = DftiGetValueDM(handle, param, value)
```

C:

```
status = DftiGetValueDM(handle, param, &value);
```

Include Files

- Fortran 90: mkl_cdft.f90
- C: mkl_cdft.h

Input Parameters

handle

The descriptor handle. Must be valid, that is, created in a call to [DftiCreateDescriptorDM](#).

param

Name of a parameter to be retrieved from the descriptor. See [Table "Retrievable Configuration Parameters"](#) for the list of available parameters.

Output Parameters

<code>value</code>	Value of the parameter.
--------------------	-------------------------

Description

This function gets the configuration value of one particular configuration parameter. The configuration parameter is one of the named constants listed in the table below, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type. Possible values of the named constants can be found in [Table "Configuration Parameters"](#) and relevant subsections of the [Configuration Settings](#) section.

Retrievable Configuration Parameters

Parameter Name	Data Type	Description
<code>DFTI_PRECISION</code>	Named constant	Precision of computation, input data and output data.
<code>DFTI_DIMENSION</code>	Integer scalar	Dimension of the transform
<code>DFTI_LENGTHS</code>	Array of integer values	Array of lengths of the transform. Number of lengths corresponds to the dimension of the transform.
<code>DFTI_FORWARD_SCALE</code>	Floating-point scalar	Scale factor of forward transform.
<code>DFTI_BACKWARD_SCALE</code>	Floating-point scalar	Scale factor of backward transform.
<code>DFTI_PLACEMENT</code>	Named constant	Placement of the computation result.
<code>DFTI_COMMIT_STATUS</code>	Named constant	Shows whether descriptor has been committed.
<code>DFTI_FORWARD_DOMAIN</code>	Named constant	Forward domain of transforms, has the value of <code>DFTI_COMPLEX</code> or <code>DFTI_REAL</code> .
<code>DFTI_ORDERING</code>	Named constant	Scrambling of data order.
<code>CDFT_MPI_COMM</code>	Type of MPI communicator	MPI communicator used for transforms.
<code>CDFT_LOCAL_SIZE</code>	Integer scalar	Necessary size of input, output, and buffer arrays in data elements.
<code>CDFT_LOCAL_X_START</code>	Integer scalar	Row/element number of the global array that corresponds to the first row/element of the local array. For more information, see Distributing Data among Processes .
<code>CDFT_LOCAL_NX</code>	Integer scalar	The number of rows/elements of the global array stored in the local array. For more information, see Distributing Data among Processes .
<code>CDFT_LOCAL_OUT_X_START</code>	Integer scalar	Element number of the appropriate global array that corresponds to the first element of the input or output local array in a 1D case. For details, see Distributing Data among Processes .

Parameter Name	Data Type	Description
CDFT_LOCAL_OUT_NX	Integer scalar	The number of elements of the appropriate global array that are stored in the input or output local array in a 1D case. For details, see Distributing Data among Processes .

Return Values

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the [Error Codes](#) section).

Interface

```

! Fortran interface
INTERFACE DftiGetValueDM
    INTEGER(4) FUNCTION DftiGetValueDM(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p, v
    END FUNCTION
    INTEGER(4) FUNCTION DftiGetValueDMar(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p, v(*)
    END FUNCTION
    INTEGER(4) FUNCTION DftiGetValueDMd(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p
        REAL(8) :: v
    END FUNCTION
    INTEGER(4) FUNCTION DftiGetValueDMs(h, p, v)
        TYPE(DFTI_DESCRIPTOR_DM), POINTER :: h
        INTEGER(4) :: p
        REAL(4) :: v
    END FUNCTION
END INTERFACE

```

Prototype

```

/* C/C++ prototype */
MKL_LONG DftiGetValueDM(DFTI_DESCRIPTOR_DM_HANDLE handle, int param,...);

```

Error Codes

All the cluster FFT functions return an integer value denoting the status of the operation. These values are identified by named constants. Each function returns `DFTI_NO_ERROR` if no errors were encountered during execution. Otherwise, a function generates an error code. In addition to FFT error codes, the cluster FFT interface has its own ones. Named constants specific to the cluster FFT interface have prefix "CDFT" in names. [Table "Error Codes that Cluster FFT Functions Return"](#) lists error codes that the cluster FFT functions may return.

Error Codes that Cluster FFT Functions Return

Named Constants	Comments
<code>DFTI_NO_ERROR</code>	No error.

Named Constants	Comments
DFTI_MEMORY_ERROR	Usually associated with memory allocation.
DFTI_INVALID_CONFIGURATION	Invalid settings of one or more configuration parameters.
DFTI_INCONSISTENT_CONFIGURATION	Inconsistent configuration or input parameters.
DFTI_NUMBER_OF_THREADS_ERROR	Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function).
DFTI_MULTITHREADED_ERROR	Usually associated with a value that OMP routines return in case of errors.
DFTI_BAD_DESCRIPTOR	Descriptor is unusable for computation.
DFTI_UNIMPLEMENTED	Unimplemented legitimate settings; implementation dependent.
DFTI_MKL_INTERNAL_ERROR	Internal library error.
DFTI_1D_LENGTH_EXCEEDS_INT32	Length of one of dimensions exceeds $2^{32} - 1$ (4 bytes).
CDFT_SPREAD_ERROR	Data cannot be distributed (For more information, see Distributing Data among Processes .)
CDFT_MPI_ERROR	MPI error. Occurs when calling MPI.

PBLAS Routines

This chapter describes the Intel® Math Kernel Library implementation of the PBLAS (Parallel Basic Linear Algebra Subprograms) routines from the ScaLAPACK package for distributed-memory architecture. PBLAS is intended for using in vector-vector, matrix-vector, and matrix-matrix operations to simplify the parallelization of linear codes. The design of PBLAS is as consistent as possible with that of the BLAS. The routine descriptions are arranged in several sections according to the PBLAS level of operation:

- [PBLAS Level 1 Routines](#) (distributed vector-vector operations)
- [PBLAS Level 2 Routines](#) (distributed matrix-vector operations)
- [PBLAS Level 3 Routines](#) (distributed matrix-matrix operations)

Each section presents the routine and function group descriptions in alphabetical order by the routine group name; for example, the `p?asum` group, the `p?axpy` group. The question mark in the group name corresponds to a character indicating the data type (`s`, `d`, `c`, and `z` or their combination); see [Routine Naming Conventions](#).

NOTE

PBLAS routines are provided only with Intel® MKL versions for Linux* and Windows* OSs.

Generally, PBLAS runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of PBLAS optimized for the target architecture. The Intel MKL version of PBLAS is optimized for Intel® processors. For the detailed system and environment requirements see [Intel® MKL Release Notes](#) and [Intel® MKL User's Guide](#).

For full reference on PBLAS routines and related information, see http://www.netlib.org/scalapack/html/pblas_qref.html.

Optimization Notice

<p>Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.</p>
--

Notice revision #20110804

Overview

The model of the computing environment for PBLAS is represented as a one-dimensional array of processes or also a two-dimensional process grid. To use PBLAS, all global matrices or vectors must be distributed on this array or grid prior to calling the PBLAS routines.

PBLAS uses the two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, as well as gives the opportunity to use PBLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global array and its corresponding process and memory location is contained in the so called *array descriptor* associated with each global array. [Table "Content of the array descriptor for dense matrices"](#) gives an example of an array descriptor structure.

Content of Array Descriptor for Dense Matrices

Array Element #	Name	Definition
1	<i>dtype</i>	Descriptor type (=1 for dense matrices)
2	<i>ctxt</i>	BLACS context handle for the process grid
3	<i>m</i>	Number of rows in the global array
4	<i>n</i>	Number of columns in the global array
5	<i>mb</i>	Row blocking factor
6	<i>nb</i>	Column blocking factor
7	<i>rsrc</i>	Process row over which the first row of the global array is distributed
8	<i>csrc</i>	Process column over which the first column of the global array is distributed
9	<i>lld</i>	Leading dimension of the local array

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by *LOCr()* and *LOCc()*, respectively. To compute these numbers, you can use the ScaLAPACK tool routine *numroc*.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix of the global matrix *A*, which is contained in the global subarray *sub(A)*, defined by the following 6 values (for dense matrices):

<i>m</i>	The number of rows of <i>sub(A)</i>
<i>n</i>	The number of columns of <i>sub(A)</i>
<i>a</i>	A pointer to the local array containing the entire global array <i>A</i>
<i>ia</i>	The row index of <i>sub(A)</i> in the global array
<i>ja</i>	The column index of <i>sub(A)</i> in the global array
<i>desca</i>	The array descriptor for the global array <i>A</i>

Intel MKL provides the PBLAS routines with interface similar to the interface used in the Netlib PBLAS (see http://www.netlib.org/scalapack/html/pblas_qref.html).

Routine Naming Conventions

The naming convention for PBLAS routines is similar to that used for BLAS routines (see [Routine Naming Conventions in Chapter 2](#)). A general rule is that each routine name in PBLAS, which has a BLAS equivalent, is simply the BLAS name prefixed by initial letter *p* that stands for "parallel".

The Intel MKL PBLAS routine names have the following structure:

```
p <character> <name> <mod> ( )
```

The *<character>* field indicates the Fortran data type:

<i>s</i>	real, single precision
<i>c</i>	complex, single precision
<i>d</i>	real, double precision
<i>z</i>	complex, double precision
<i>i</i>	integer

Some routines and functions can have combined character codes, such as *sc* or *dz*.

For example, the function *pscasum* uses a complex input array and returns a real value.

The *<name>* field, in PBLAS level 1, indicates the operation type. For example, the PBLAS level 1 routines p?dot, p?swap, p?copy compute a vector dot product, vector swap, and a copy vector, respectively.

In PBLAS level 2 and 3, *<name>* reflects the matrix argument type:

ge	general matrix
sy	symmetric matrix
he	Hermitian matrix
tr	triangular matrix

In PBLAS level 3, the *<name>=tran* indicates the transposition of the matrix.

The *<mod>* field, if present, provides additional details of the operation. The PBLAS level 1 names can have the following characters in the *<mod>* field:

c	conjugated vector
u	unconjugated vector

The PBLAS level 2 names can have the following additional characters in the *<mod>* field:

mv	matrix-vector product
sv	solving a system of linear equations with matrix-vector operations
r	rank-1 update of a matrix
r2	rank-2 update of a matrix.

The PBLAS level 3 names can have the following additional characters in the *<mod>* field:

mm	matrix-matrix product
sm	solving a system of linear equations with matrix-matrix operations
rk	rank- <i>k</i> update of a matrix
r2k	rank-2 <i>k</i> update of a matrix.

The examples below show how to interpret PBLAS routine names:

pddot	<p> <d> <dot>: double-precision real distributed vector-vector dot product
pcdotc	<p> <c> <dot> <c>: complex distributed vector-vector dot product, conjugated
pscasum	<p> <sc> <asum>: sum of magnitudes of distributed vector elements, single precision real output and single precision complex input
pcdotu	<p> <c> <dot> <u>: distributed vector-vector dot product, unconjugated, complex
psgemv	<p> <s> <ge> <mv>: distributed matrix-vector product, general matrix, single precision
pztrmm	<p> <z> <tr> <mm>: distributed matrix-matrix product, triangular matrix, double-precision complex.

PBLAS Level 1 Routines

PBLAS Level 1 includes routines and functions that perform distributed vector-vector operations. Table "PBLAS Level 1 Routine Groups and Their Data Types" lists the PBLAS Level 1 routine groups and the data types associated with them.

PBLAS Level 1 Routine Groups and Their Data Types

Routine or Function Group	Data Types	Description
p?amax	s, d, c, z	Calculates an index of the distributed vector element with maximum absolute value
p?asum	s, d, sc, dz	Calculates sum of magnitudes of a distributed vector
p?axpy	s, d, c, z	Calculates distributed vector-scalar product
p?copy	s, d, c, z	Copies a distributed vector
p?dot	s, d	Calculates a dot product of two distributed real vectors
p?dotc	c, z	Calculates a dot product of two distributed complex vectors, one of them is conjugated
p?dotu	c, z	Calculates a dot product of two distributed complex vectors
p?nrm2	s, d, sc, dz	Calculates the 2-norm (Euclidean norm) of a distributed vector
p?scal	s, d, c, z, cs, zd	Calculates a product of a distributed vector by a scalar
p?swap	s, d, c, z	Swaps two distributed vectors

p?amax

Computes the global index of the element of a distributed vector with maximum absolute value.

Syntax

Fortran:

```
call psamax(n, amax, indx, x, ix, jx, descx, incx)
call pdamax(n, amax, indx, x, ix, jx, descx, incx)
call pcamax(n, amax, indx, x, ix, jx, descx, incx)
call pzamax(n, amax, indx, x, ix, jx, descx, incx)
```

C:

```
void psamax (MKL_INT *n , float *amax , MKL_INT *indx , float *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx );
void pdamax (MKL_INT *n , double *amax , MKL_INT *indx , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx );
void pcamax (MKL_INT *n , float *amax , MKL_INT *indx , float *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx );
void pzamax (MKL_INT *n , double *amax , MKL_INT *indx , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx );
```

Include Files

- C: mkl_pblas.h

Description

The functions p?amax compute global index of the maximum element in absolute value of a distributed vector sub(x),

where sub(x) denotes $x(ix, jx:jx+n-1)$ if $incx=m_x$, and $x(ix: ix+n-1, jx)$ if $incx= 1$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of distributed vector sub(x), $n \geq 0$.
<i>x</i>	(local) REAL for psamax DOUBLE PRECISION for pdamax COMPLEX for pcamax DOUBLE COMPLEX for pzamax Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector sub(x).
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix sub(X), respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X.
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of sub(x). Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.

Output Parameters

<i>amax</i>	(global) REAL for psamax . DOUBLE PRECISION for pdamax. COMPLEX for pcamax. DOUBLE COMPLEX for pzamax. Maximum absolute value (magnitude) of elements of the distributed vector only in its scope.
<i>indx</i>	(global) INTEGER. The global index of the maximum element in absolute value of the distributed vector sub(x) only in its scope.

p?asum

Computes the sum of magnitudes of elements of a distributed vector.

Syntax

Fortran:

```
call psasum(n, asum, x, ix, jx, descx, incx)
```

```
call pscasum(n, asum, x, ix, jx, descx, incx)
call pdasum(n, asum, x, ix, jx, descx, incx)
call pdzasum(n, asum, x, ix, jx, descx, incx)
```

C:

```
void psasum (MKL_INT *n , float *asum , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pdasum (MKL_INT *n , double *asum , double *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
void pscasum (MKL_INT *n , float *asum , float *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
void pdzasum (MKL_INT *n , double *asum , double *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
```

Include Files

- C: mkl_pblas.h

Description

The functions p?asum compute the sum of the magnitudes of elements of a distributed vector $\text{sub}(x)$, where $\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix: ix+n-1, jx)$ if $incx= 1$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of distributed vector $\text{sub}(x)$, $n \geq 0$.
<i>x</i>	(local) REAL for psasum DOUBLE PRECISION for pdasum COMPLEX for pscasum DOUBLE COMPLEX for pdzasum Array, size $(jx-1)*m_x + ix+(n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix</i> , <i>jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(X)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

Output Parameters

<i>asum</i>	(local) REAL for psasum and pscasum. DOUBLE PRECISION for pdasum and pdzasum
-------------	---

Contains the sum of magnitudes of elements of the distributed vector only in its scope.

p?axpy

Computes a distributed vector-scalar product and adds the result to a distributed vector.

Syntax

Fortran:

```
call psaxpy(n, a, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pdaxpy(n, a, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pcaxpy(n, a, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzaxpy(n, a, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

C:

```
void psaxpy (MKL_INT *n , float *a , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy ,
MKL_INT *incy );
void pdaxpy (MKL_INT *n , double *a , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy ,
MKL_INT *incy );
void pcaxpy (MKL_INT *n , float *a , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy ,
MKL_INT *incy );
void pzaxpy (MKL_INT *n , double *a , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy ,
MKL_INT *incy );
```

Include Files

- C: mkl_pblas.h

Description

The p?axpy routines perform the following operation with distributed vectors:

```
sub(y) := sub(y) + a*sub(x)
```

where:

a is a scalar;

sub(x) and *sub(y)* are *n*-element distributed vectors.

sub(x) denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix: ix+n-1, jx)$ if $incx= 1$;

sub(y) denotes $Y(iy, jy:jy+n-1)$ if $incy=m_y$, and $Y(iy: iy+n-1, jy)$ if $incy= 1$.

Input Parameters

n (global) INTEGER. The length of distributed vectors, $n \geq 0$.

a (local) REAL for psaxpy

DOUBLE PRECISION for pdaxpy

COMPLEX for pcaxpy

DOUBLE COMPLEX for pzaxpy

Specifies the scalar a .

x

(local) REAL for psaxpy

DOUBLE PRECISION for pdaxpy

COMPLEX for pcaxpy

DOUBLE COMPLEX for pzaxpy

Array, size $(jx-1)*m_x + ix + (n-1)*\text{abs}(incx)$.

This array contains the entries of the distributed vector $\text{sub}(x)$.

ix, jx

(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.

$descx$

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .

$incx$

(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

y

(local) REAL for psaxpy

DOUBLE PRECISION for pdaxpy

COMPLEX for pcaxpy

DOUBLE COMPLEX for pzaxpy

Array, size $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$.

This array contains the entries of the distributed vector $\text{sub}(y)$.

iy, jy

(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.

$descy$

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .

$incy$

(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . $incy$ must not be zero.

Output Parameters

y

Overwritten by $\text{sub}(y) := \text{sub}(y) + a * \text{sub}(x)$.

p?copy

Copies one distributed vector to another vector.

Syntax

Fortran:

```
call pscopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pdcopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pccopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzcopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call picopy(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

C:

```
void pscopy (MKL_INT *n , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_INT
*incy );
void pdcopy (MKL_INT *n , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_INT
*incy );
void pccopy (MKL_INT *n , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_INT
*incy );
void pzcopy (MKL_INT *n , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_INT
*incy );
void picopy (MKL_INT *n , MKL_INT *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , MKL_INT *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_INT
*incy );
```

Include Files

- C: mkl_pblas.h

Description

The p?copy routines perform a copy operation with distributed vectors defined as

```
sub(y) = sub(x),
```

where sub(x) and sub(y) are n -element distributed vectors.

sub(x) denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix: ix+n-1, jx)$ if $incx= 1$;

sub(y) denotes $Y(iy, jy:jy+n-1)$ if $incy=m_y$, and $Y(iy: iy+n-1, jy)$ if $incy= 1$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of distributed vectors, $n \geq 0$.
<i>x</i>	(local) REAL for pscopy DOUBLE PRECISION for pdcopy COMPLEX for pccopy DOUBLE COMPLEX for pzcopy INTEGER for picopy Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$.

This array contains the entries of the distributed vector `sub(x)`.

`ix, jx`

(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix `sub(x)`, respectively.

`descx`

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .

`incx`

(global) INTEGER. Specifies the increment for the elements of `sub(x)`. Only two values are supported, namely 1 and m_x . `incx` must not be zero.

`y`

(local) REAL for `pscopy`

DOUBLE PRECISION for `pdcopy`

COMPLEX for `pccopy`

DOUBLE COMPLEX for `pzccopy`

INTEGER for `picopy`

Array, size $(jy-1) * m_y + iy + (n-1) * \text{abs}(incy)$.

This array contains the entries of the distributed vector `sub(y)`.

`iy, jy`

(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix `sub(y)`, respectively.

`descy`

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .

`incy`

(global) INTEGER. Specifies the increment for the elements of `sub(y)`. Only two values are supported, namely 1 and m_y . `incy` must not be zero.

Output Parameters

`y`

Overwritten with the distributed vector `sub(x)`.

p?dot

Computes the dot product of two distributed real vectors.

Syntax

Fortran:

```
call psdot(n, dot, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pddot(n, dot, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

C:

```
void psdot (MKL_INT *n , float *dot , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy ,
MKL_INT *incy );
```

```
void pddot (MKL_INT *n , double *dot , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy ,
MKL_INT *incy );
```

Include Files

- C: mkl_pblas.h

Description

The ?dot functions compute the dot product *dot* of two distributed real vectors defined as

```
dot = sub(x)'*sub(y)
```

where *sub(x)* and *sub(y)* are *n*-element distributed vectors.

sub(x) denotes $X(ix, jx:jx+n-1)$ if *incx=m_x*, and $X(ix: ix+n-1, jx)$ if *incx= 1*;

sub(y) denotes $Y(iy, jy:jy+n-1)$ if *incy=m_y*, and $Y(iy: iy+n-1, jy)$ if *incy= 1*.

Input Parameters

<i>n</i>	(global) INTEGER. The length of distributed vectors, $n \geq 0$.
<i>x</i>	<p>(local) REAL for psdot</p> <p>DOUBLE PRECISION for pddot</p> <p>Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$.</p> <p>This array contains the entries of the distributed vector <i>sub(x)</i> .</p>
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <i>sub(X)</i> , respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(x)</i> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.
<i>y</i>	<p>(local)REAL for psdot</p> <p>DOUBLE PRECISION for pddot</p> <p>Array, size $(jy-1)*m_y + iy+(n-1)*abs(incy)$.</p> <p>This array contains the entries of the distributed vector <i>sub(y)</i> .</p>
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <i>sub(Y)</i> , respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(y)</i> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.

Output Parameters

`dot` (local) REAL for psdot
 DOUBLE PRECISION for pddot
 Dot product of `sub(x)` and `sub(y)` only in their scope.

p?dotc

Computes the dot product of two distributed complex vectors, one of them is conjugated.

Syntax

Fortran:

```
call pcdotc(n, dotc, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzdotc(n, dotc, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

C:

```
void pcdotc (MKL_INT *n , float *dotc , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy ,
MKL_INT *incy );
void pzdotc (MKL_INT *n , double *dotc , double *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy , MKL_INT
*descy , MKL_INT *incy );
```

Include Files

- C: mkl_pblas.h

Description

The `p?dotc` functions compute the dot product `dotc` of two distributed vectors, with one vector conjugated:

```
dotc = conjg(sub(x)')*sub(y)
```

where `sub(x)` and `sub(y)` are n -element distributed vectors.

`sub(x)` denotes $X(ix, jx:jx+n-1)$ if `incx=m_x`, and $X(ix: ix+n-1, jx)$ if `incx= 1`;
`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if `incy=m_y`, and $Y(iy: iy+n-1, jy)$ if `incy= 1`.

Input Parameters

`n` (global) INTEGER. The length of distributed vectors, $n \geq 0$.
`x` (local) COMPLEX for pcdotc
 DOUBLE COMPLEX for pzdotc
 Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$.
 This array contains the entries of the distributed vector `sub(x)`.
`ix, jx` (global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix `sub(x)`, respectively.

<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(x)</i> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.
<i>y</i>	(local) COMPLEX for pcdotc DOUBLE COMPLEX for pzdotc Array, size $(jy-1) * m_y + iy + (n-1) * \text{abs}(incy)$. This array contains the entries of the distributed vector <i>sub(y)</i> .
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <i>sub(Y)</i> , respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(y)</i> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.

Output Parameters

<i>dotc</i>	(local) COMPLEX for pcdotc DOUBLE COMPLEX for pzdotc Dot product of <i>sub(x)</i> and <i>sub(y)</i> only in their scope.
-------------	--

p?dotu

Computes the dot product of two distributed complex vectors.

Syntax

Fortran:

```
call pcdotu(n, dotu, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzdotu(n, dotu, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

C:

```
void pcdotu (MKL_INT *n , float *dotu , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy ,
MKL_INT *incy );
void pzdotu (MKL_INT *n , double *dotu , double *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy , MKL_INT
*descy , MKL_INT *incy );
```

Include Files

- C: mkl_pblas.h

Description

The p?dotu functions compute the dot product $dotu$ of two distributed vectors defined as

```
dotu = sub(x)' * sub(y)
```

where $sub(x)$ and $sub(y)$ are n -element distributed vectors.

$sub(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix: ix+n-1, jx)$ if $incx= 1$;

$sub(y)$ denotes $Y(iy, jy:jy+n-1)$ if $incy=m_y$, and $Y(iy: iy+n-1, jy)$ if $incy= 1$.

Input Parameters

n	(global) INTEGER. The length of distributed vectors, $n \geq 0$.
x	(local) COMPLEX for pcdotu DOUBLE COMPLEX for pzdotu Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector $sub(x)$.
ix, jx	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $sub(X)$, respectively.
$descx$	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
$incx$	(global) INTEGER. Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.
y	(local) COMPLEX for pcdotu DOUBLE COMPLEX for pzdotu Array, size $(jy-1)*m_y + iy+(n-1)*abs(incy)$. This array contains the entries of the distributed vector $sub(y)$.
iy, jy	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $sub(Y)$, respectively.
$descy$	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
$incy$	(global) INTEGER. Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and m_y . $incy$ must not be zero.

Output Parameters

$dotu$	(local) COMPLEX for pcdotu DOUBLE COMPLEX for pzdotu Dot product of $sub(x)$ and $sub(y)$ only in their scope.
--------	--

p?nrm2

Computes the Euclidean norm of a distributed vector.

Syntax

Fortran:

```
call psnrm2(n, norm2, x, ix, jx, descx, incx)
call pdnrm2(n, norm2, x, ix, jx, descx, incx)
call pscnrm2(n, norm2, x, ix, jx, descx, incx)
call pdznrm2(n, norm2, x, ix, jx, descx, incx)
```

C:

```
void psnrm2 (MKL_INT *n , float *norm2 , float *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
void pdnrm2 (MKL_INT *n , double *norm2 , double *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
void pscnrm2 (MKL_INT *n , float *norm2 , float *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
void pdznrm2 (MKL_INT *n , double *norm2 , double *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx );
```

Include Files

- C: mkl_pblas.h

Description

The p?nrm2 functions compute the Euclidean norm of a distributed vector $\text{sub}(x)$, where $\text{sub}(x)$ is an n -element distributed vector.

$\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix: ix+n-1, jx)$ if $incx= 1$.

Input Parameters

n	(global) INTEGER. The length of distributed vector $\text{sub}(x)$, $n \geq 0$.
x	(local) REAL for psnrm2 DOUBLE PRECISION for pdnrm2 COMPLEX for pscnrm2 DOUBLE COMPLEX for pdznrm2 Array, size $(jx-1) * m_x + ix + (n-1) * \text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
ix, jx	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(X)$, respectively.
$descx$	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .

incx (global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

Output Parameters

norm2 (local) REAL for psnrm2 and pscfrm2 .
 DOUBLE PRECISION for pdnrm2 and pdznm2

Contains the Euclidean norm of a distributed vector only in its scope.

p?scal

Computes a product of a distributed vector by a scalar.

Syntax

Fortran:

```
call psscal(n, a, x, ix, jx, descx, incx)
call pdscal(n, a, x, ix, jx, descx, incx)
call pcscal(n, a, x, ix, jx, descx, incx)
call pzscal(n, a, x, ix, jx, descx, incx)
call pcscal(n, a, x, ix, jx, descx, incx)
call pdscal(n, a, x, ix, jx, descx, incx)
```

C:

```
void psscal (MKL_INT *n , float *a , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pdscal (MKL_INT *n , double *a , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pcscal (MKL_INT *n , float *a , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pzscal (MKL_INT *n , double *a , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pcscal (MKL_INT *n , float *a , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pdscal (MKL_INT *n , double *a , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
```

Include Files

- C: mkl_pblas.h

Description

The $p?\text{scal}$ routines multiplies a n -element distributed vector $\text{sub}(x)$ by the scalar a :

$\text{sub}(x) = a * \text{sub}(x)$,

where $\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx=m_x$, and $X(ix: ix+n-1, jx)$ if $incx= 1$.

Input Parameters

<i>n</i>	(global) INTEGER. The length of distributed vector <code>sub(x)</code> , $n \geq 0$.
<i>a</i>	(global) REAL for <code>psscal</code> and <code>pcsscal</code> DOUBLE PRECISION for <code>pdscal</code> and <code>pdscal</code> COMPLEX for <code>pcscal</code> DOUBLE COMPLEX for <code>pzscal</code> Specifies the scalar <i>a</i> .
<i>x</i>	(local) REAL for <code>psscal</code> DOUBLE PRECISION for <code>pdscal</code> COMPLEX for <code>pcscal</code> and <code>pcsscal</code> DOUBLE COMPLEX for <code>pzscal</code> and <code>pdscal</code> Array, size $(jx-1)*m_x + ix + (n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <code>sub(X)</code> , respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.

Output Parameters

<i>x</i>	Overwritten by the updated distributed vector <code>sub(x)</code>
----------	---

p?swap

Swaps two distributed vectors.

Syntax

Fortran:

```
call psswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pdswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pcswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
call pzswap(n, x, ix, jx, descx, incx, y, iy, jy, descy, incy)
```

C:

```
void psswap (MKL_INT *n , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_INT
*incy );
```

```

void pdswap (MKL_INT *n , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_INT
*incy );

void pcswap (MKL_INT *n , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_INT
*incy );

void pzswap (MKL_INT *n , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_INT
*incy );

```

Include Files

- C: mkl_pblas.h

Description

Given two distributed vectors `sub(x)` and `sub(y)`, the p?swap routines return vectors `sub(y)` and `sub(x)` swapped, each replacing the other.

Here `sub(x)` denotes $X(ix, jx:jx+n-1)$ if `incx=m_x`, and $X(ix: ix+n-1, jx)$ if `incx= 1`;

`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if `incy=m_y`, and $Y(iy: iy+n-1, jy)$ if `incy= 1`.

Input Parameters

<code>n</code>	(global) INTEGER. The length of distributed vectors, $n \geq 0$.
<code>x</code>	(local) REAL for psswap DOUBLE PRECISION for pdswap COMPLEX for pcswap DOUBLE COMPLEX for pzswap Array, size $(jx-1)*m_x + ix+(n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<code>ix, jx</code>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix <code>sub(X)</code> , respectively.
<code>descx</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<code>incx</code>	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and <code>m_x</code> . <code>incx</code> must not be zero.
<code>y</code>	(local) REAL for psswap DOUBLE PRECISION for pdswap COMPLEX for pcswap DOUBLE COMPLEX for pzswap Array, size $(jy-1)*m_y + iy+(n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector <code>sub(y)</code> .

<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(Y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(Y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.

Output Parameters

<i>x</i>	Overwritten by distributed vector $\text{sub}(y)$.
<i>y</i>	Overwritten by distributed vector $\text{sub}(x)$.

PBLAS Level 2 Routines

This section describes PBLAS Level 2 routines, which perform distributed matrix-vector operations. [Table "PBLAS Level 2 Routine Groups and Their Data Types"](#) lists the PBLAS Level 2 routine groups and the data types associated with them.

PBLAS Level 2 Routine Groups and Their Data Types

Routine Groups	Data Types	Description
p?gemv	s, d, c, z	Matrix-vector product using a distributed general matrix
p?agemv	s, d, c, z	Matrix-vector product using absolute values for a distributed general matrix
p?ger	s, d	Rank-1 update of a distributed general matrix
p?gerc	c, z	Rank-1 update (conjugated) of a distributed general matrix
p?geru	c, z	Rank-1 update (unconjugated) of a distributed general matrix
p?hemv	c, z	Matrix-vector product using a distributed Hermitian matrix
p?ahemv	c, z	Matrix-vector product using absolute values for a distributed Hermitian matrix
p?her	c, z	Rank-1 update of a distributed Hermitian matrix
p?her2	c, z	Rank-2 update of a distributed Hermitian matrix
p?symv	s, d	Matrix-vector product using a distributed symmetric matrix
p?asymv	s, d	Matrix-vector product using absolute values for a distributed symmetric matrix
p?sy	s, d	Rank-1 update of a distributed symmetric matrix
p?sy2	s, d	Rank-2 update of a distributed symmetric matrix
p?trmv	s, d, c, z	Distributed matrix-vector product using a triangular matrix

Routine Groups	Data Types	Description
p?atrmv	s, d, c, z	Distributed matrix-vector product using absolute values for a triangular matrix
p?trsv	s, d, c, z	Solves a system of linear equations whose coefficients are in a distributed triangular matrix

p?gemv

Computes a distributed matrix-vector product using a general matrix.

Syntax

Fortran:

```
call psgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
            iy, jy, descy, incy)

call pdgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
            iy, jy, descy, incy)

call pcgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
            iy, jy, descy, incy)

call pzgemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
            iy, jy, descy, incy)
```

C:

```
void psgemv (char *trans , MKL_INT *m , MKL_INT *n , float *alpha , float *a , MKL_INT
             *ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
             *descx , MKL_INT *incx , float *beta , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT
             *descy , MKL_INT *incy );

void pdgemv (char *trans , MKL_INT *m , MKL_INT *n , double *alpha , double *a ,
             MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT *jx ,
             MKL_INT *descx , MKL_INT *incx , double *beta , double *y , MKL_INT *iy , MKL_INT
             *jy , MKL_INT *descy , MKL_INT *incy );

void pcgemv (char *trans , MKL_INT *m , MKL_INT *n , float *alpha , float *a , MKL_INT
             *ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
             *descx , MKL_INT *incx , float *beta , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT
             *descy , MKL_INT *incy );

void pzgemv (char *trans , MKL_INT *m , MKL_INT *n , double *alpha , double *a ,
             MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT *jx ,
             MKL_INT *descx , MKL_INT *incx , double *beta , double *y , MKL_INT *iy , MKL_INT
             *jy , MKL_INT *descy , MKL_INT *incy );
```

Include Files

- C: mkl_pblas.h

Description

The p?gemv routines perform a distributed matrix-vector operation defined as

```
sub(y) := alpha*sub(A)*sub(x) + beta*sub(y),
```

or

```
sub(y) := alpha*sub(A)'*sub(x) + beta*sub(y),
```

or

```
sub(y) := alpha*conjg(sub(A)')*sub(x) + beta*sub(y),
```

where

alpha and *beta* are scalars,

sub(A) is a m -by- n submatrix, *sub(A)* = $A(ia:ia+m-1, ja:ja+n-1)$,

sub(x) and *sub(y)* are subvectors.

When *trans* = 'N' or 'n', *sub(x)* denotes $X(ix, jx:jx+n-1)$ if *incx* = *m_x*, and $X(ix: ix+n-1, jx)$ if *incx* = 1, *sub(y)* denotes $Y(iy, jy:jy+m-1)$ if *incy* = *m_y*, and $Y(iy: iy+m-1, jy)$ if *incy* = 1.

When *trans* = 'T' or 't', or 'C', or 'c', *sub(x)* denotes $X(ix, jx:jx+m-1)$ if *incx* = *m_x*, and $X(ix: ix+m-1, jx)$ if *incx* = 1, *sub(y)* denotes $Y(iy, jy:jy+n-1)$ if *incy* = *m_y*, and $Y(iy: iy+m-1, jy)$ if *incy* = 1.

Input Parameters

trans

(global) CHARACTER*1. Specifies the operation:

```
if trans= 'N' or 'n', then sub(y) := alpha*sub(A)'*sub(x) +
beta*sub(y);

if trans= 'T' or 't', then sub(y) := alpha*sub(A)'*sub(x) +
beta*sub(y);

if trans= 'C' or 'c', then sub(y) := alpha*conjg(subA)')*sub(x) +
beta*sub(y).
```

m

(global) INTEGER. Specifies the number of rows of the distributed matrix *sub(A)*, $m \geq 0$.

n

(global) INTEGER. Specifies the number of columns of the distributed matrix *sub(A)*, $n \geq 0$.

alpha

(global) REAL for psgemv

DOUBLE PRECISION for pdgemv

COMPLEX for pcgemv

DOUBLE COMPLEX for pzgemv

Specifies the scalar *alpha*.

a

(local) REAL for psgemv

DOUBLE PRECISION for pdgemv

COMPLEX for pcgemv

DOUBLE COMPLEX for pzgemv

Array, size (*lld_a*, LOC_q(*ja*+*n*-1)). Before entry this array must contain the local pieces of the distributed matrix *sub(A)*.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>descA</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>x</i>	(local)REAL for psgemv DOUBLE PRECISION for pdgemv COMPLEX for pcgemv DOUBLE COMPLEX for pzgemv Array, size $(jx-1)*m_x + ix+(n-1)*\text{abs}(incx)$ when $\text{trans} = 'N'$ or ' n ', and $(jx-1)*m_x + ix+(m-1)*\text{abs}(incx)$ otherwise. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descX</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>beta</i>	(global)REAL for psgemv DOUBLE PRECISION for pdgemv COMPLEX for pcgemv DOUBLE COMPLEX for pzgemv Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then $\text{sub}(y)$ need not be set on input.
<i>y</i>	(local)REAL for psgemv DOUBLE PRECISION for pdgemv COMPLEX for pcgemv DOUBLE COMPLEX for pzgemv Array, size $(jy-1)*m_y + iy+(m-1)*\text{abs}(incy)$ when $\text{trans} = 'N'$ or ' n ', and $(jy-1)*m_y + iy+(n-1)*\text{abs}(incy)$ otherwise. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
<i>descY</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .

incy (global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . *incy* must not be zero.

Output Parameters

y Overwritten by the updated distributed vector $\text{sub}(y)$.

p?agemv

Computes a distributed matrix-vector product using absolute values for a general matrix.

Syntax

Fortran:

```
call psagemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
iy, jy, descy, incy)

call pdagemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
iy, jy, descy, incy)

call pcagemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
iy, jy, descy, incy)

call pzagemv(trans, m, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y,
iy, jy, descy, incy)
```

C:

```
void psagemv (char *trans , MKL_INT *m , MKL_INT *n , float *alpha , float *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx , float *beta , float *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , MKL_INT *incy );

void pdagemv (char *trans , MKL_INT *m , MKL_INT *n , double *alpha , double *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx , double *beta , double *y , MKL_INT *iy , MKL_INT
*jy , MKL_INT *descy , MKL_INT *incy );

void pcagemv (char *trans , MKL_INT *m , MKL_INT *n , float *alpha , float *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx , float *beta , float *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , MKL_INT *incy );

void pzagemv (char *trans , MKL_INT *m , MKL_INT *n , double *alpha , double *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT *jx ,
MKL_INT *descx , MKL_INT *incx , double *beta , double *y , MKL_INT *iy , MKL_INT
*jy , MKL_INT *descy , MKL_INT *incy );
```

Include Files

- C: mkl_pblas.h

Description

The p?agemv routines perform a distributed matrix-vector operation defined as

```
sub(y) := abs(alpha)*abs(sub(A'))*abs(sub(x)) + abs(beta*sub(y)),
```

or

```
sub(y) := abs(alpha)*abs(sub(A'))*abs(sub(x)) + abs(beta*sub(y)),
```

or

```
sub(y) := abs(alpha)*abs(conjg(sub(A')))*abs(sub(x)) + abs(beta*sub(y)),
```

where

alpha and *beta* are scalars,

sub(A) is a m -by- n submatrix, *sub(A)* = $A(ia:ia+m-1, ja:ja+n-1)$,

sub(x) and *sub(y)* are subvectors.

When *trans* = 'N' or 'n',

sub(x) denotes $X(ix:ix, jx:jx+n-1)$ if *incx* = *m_x*, and

$X(ix:ix+n-1, jx:jx)$ if *incx* = 1,

sub(y) denotes $Y(iy:iy, jy:jy+m-1)$ if *incy* = *m_y*, and

$Y(iy:iy+m-1, jy:jy)$ if *incy* = 1.

When *trans* = 'T' or 't', or 'C', or 'c',

sub(x) denotes $X(ix:ix, jx:jx+m-1)$ if *incx* = *m_x*, and

$X(ix:ix+m-1, jx:jx)$ if *incx* = 1,

sub(y) denotes $Y(iy:iy, jy:jy+n-1)$ if *incy* = *m_y*, and

$Y(iy:iy+n-1, jy:jy)$ if *incy* = 1.

Input Parameters

trans

(global) CHARACTER*1. Specifies the operation:

```
if trans= 'N' or 'n', then sub(y) := |alpha|*|sub(A)|*|sub(x)| + |beta*sub(y)|
```

```
if trans= 'T' or 't', then sub(y) := |alpha|*|sub(A')|*|sub(x)| + |beta*sub(y)|
```

```
if trans= 'C' or 'c', then sub(y) := |alpha|*|sub(A')|*|sub(x)| + |beta*sub(y)|.
```

m

(global) INTEGER. Specifies the number of rows of the distributed matrix *sub(A)*, $m \geq 0$.

n

(global) INTEGER. Specifies the number of columns of the distributed matrix *sub(A)*, $n \geq 0$.

alpha

(global) REAL for psagemv

DOUBLE PRECISION for pdagemv

COMPLEX for pcagemv

DOUBLE COMPLEX for pzagemv

Specifies the scalar *alpha*.

a

(local) REAL for psagemv

DOUBLE PRECISION for pdagemv

COMPLEX for pcagemv

DOUBLE COMPLEX for pzagemv

Array, size ($l1d_a$, LOC $q(ja+n-1)$). Before entry this array must contain the local pieces of the distributed matrix $\text{sub}(A)$.

ia, ja

(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

$desc_a$

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

x

(local)REAL for psagemv

DOUBLE PRECISION for pdagemv

COMPLEX for pcagemv

DOUBLE COMPLEX for pzagemv

Array, size $(jx-1)*m_x + ix + (n-1)*\text{abs}(incx)$ when $\text{trans} = 'N'$ or ' n ', and $(jx-1)*m_x + ix + (m-1)*\text{abs}(incx)$ otherwise.

This array contains the entries of the distributed vector $\text{sub}(x)$.

ix, jx

(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.

$desc_x$

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .

$incx$

(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

$beta$

(global)REAL for psagemv

DOUBLE PRECISION for pdagemv

COMPLEX for pcagemv

DOUBLE COMPLEX for pzagemv

Specifies the scalar β . When β is set to zero, then $\text{sub}(y)$ need not be set on input.

y

(local)REAL for psagemv

DOUBLE PRECISION for pdagemv

COMPLEX for pcagemv

DOUBLE COMPLEX for pzagemv

Array, size $(jy-1)*m_y + iy + (m-1)*\text{abs}(incy)$ when $\text{trans} = 'N'$ or ' n ', and $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$ otherwise.

This array contains the entries of the distributed vector $\text{sub}(y)$.

<i>i_y, j_y</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <i>sub(y)</i> , respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(y)</i> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.

Output Parameters

<i>Y</i>	Overwritten by the updated distributed vector <i>sub(y)</i> .
----------	---

p?ger

Performs a rank-1 update of a distributed general matrix.

Syntax

Fortran:

```
call psger(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
call pdger(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

C:

```
void psger (MKL_INT *m , MKL_INT *n , float *alpha , float *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT
*descy , MKL_INT *incy , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pdger (MKL_INT *m , MKL_INT *n , double *alpha , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , MKL_INT *incy , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca );
```

Include Files

- C: mkl_pblas.h

Description

The p?ger routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*sub(y)' + sub(A),
```

where:

alpha is a scalar,

sub(A) is a *m*-by-*n* distributed general matrix, sub(A)=A(ia:ia+m-1, ja:ja+n-1),

sub(x) is an *m*-element distributed vector, sub(y) is an *n*-element distributed vector,

sub(x) denotes X(ix, jx:jx+m-1) if incx = *m_x*, and X(ix: ix+m-1, jx) if incx = 1,

sub(y) denotes Y(iy, jy:jy+n-1) if incy = *m_y*, and Y(iy: iy+n-1, jy) if incy = 1.

Input Parameters

<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix $\text{sub}(A)$, $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	(global) REAL for psger DOUBLE REAL for pdger Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) REAL for psger DOUBLE REAL for pdger Array, size at least $(jx-1)*m_x + ix + (m-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	(local) REAL for psger DOUBLE REAL for pdger Array, size at least $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.
<i>a</i>	(local) REAL for psger DOUBLE REAL for pdger Array, size $(l1d_a, \text{LOCq}(ja+n-1))$. Before entry this array contains the local pieces of the distributed matrix $\text{sub}(A)$.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

<i>a</i>	Overwritten by the updated distributed matrix $\text{sub}(A)$.
----------	---

p?gerc

Performs a rank-1 update (conjugated) of a distributed general matrix.

Syntax

Fortran:

```
call pcgerc(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
call pzgerc(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja, desca)
```

C:

```
void pcgerc (MKL_INT *m , MKL_INT *n , float *alpha , float *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT
*descy , MKL_INT *incy , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pzgerc (MKL_INT *m , MKL_INT *n , double *alpha , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , MKL_INT *incy , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca );
```

Include Files

- C: mkl_pblas.h

Description

The p?gerc routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*conjg(sub(y))' + sub(A),
```

where:

alpha is a scalar,

sub(*A*) is a *m*-by-*n* distributed general matrix, sub(*A*) = $A(ia:ia+m-1, ja:ja+n-1)$,

sub(*x*) is an *m*-element distributed vector, sub(*y*) is an *n*-element distributed vector,

sub(*x*) denotes $X(ix, jx:jx+m-1)$ if *incx* = *m_x*, and $X(ix: ix+m-1, jx)$ if *incx* = 1,

sub(*y*) denotes $Y(iy, jy:jy+n-1)$ if *incy* = *m_y*, and $Y(iy: iy+n-1, jy)$ if *incy* = 1.

Input Parameters

<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix $\text{sub}(A)$, $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	(global) COMPLEX for <code>pcgerc</code> DOUBLE COMPLEX for <code>pzgerc</code> Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) COMPLEX for <code>pcgerc</code> DOUBLE COMPLEX for <code>pzgerc</code> Array, size at least $(jx-1)*m_x + ix + (n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	(local) COMPLEX for <code>pcgerc</code> DOUBLE COMPLEX for <code>pzgerc</code> Array, size at least $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.
<i>a</i>	(local) COMPLEX for <code>pcgerc</code> DOUBLE COMPLEX for <code>pzgerc</code> Array, size at least $(l1d_a, \text{LOCq}(ja+n-1))$. Before entry this array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

`descA` (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

`a` Overwritten by the updated distributed matrix $\text{sub}(A)$.

p?geru

Performs a rank-1 update (unconjugated) of a distributed general matrix.

Syntax

Fortran:

```
call pcgeru(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja,
descA)
```

```
call pzgeru(m, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia, ja,
descA)
```

C:

```
void pcgeru (MKL_INT *m , MKL_INT *n , float *alpha , float *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT
*descy , MKL_INT *incy , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *descA );
void pzgeru (MKL_INT *m , MKL_INT *n , double *alpha , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , MKL_INT *incy , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*descA );
```

Include Files

- C: mkl_pblas.h

Description

The p?geru routines perform a matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*sub(y)' + sub(A),
```

where:

`alpha` is a scalar,

`sub(A)` is a m -by- n distributed general matrix, `sub(A)=A(ia:ia+m-1, ja:ja+n-1)`,

`sub(x)` is an m -element distributed vector, `sub(y)` is an n -element distributed vector,

`sub(x)` denotes $X(ix, jx:jx+m-1)$ if `incx = m_x`, and $X(ix: ix+m-1, jx)$ if `incx = 1`,

`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if `incy = m_y`, and $Y(iy: iy+n-1, jy)$ if `incy = 1`.

Input Parameters

`m` (global) INTEGER. Specifies the number of rows of the distributed matrix $\text{sub}(A)$, $m \geq 0$.

<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	(global) COMPLEX for <code>pcgeru</code> DOUBLE COMPLEX for <code>pzgeru</code> Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) COMPLEX for <code>pcgeru</code> DOUBLE COMPLEX for <code>pzgeru</code> Array, size at least $(jx-1) * m_x + ix + (n-1) * \text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>y</i>	(local) COMPLEX for <code>pcgeru</code> DOUBLE COMPLEX for <code>pzgeru</code> Array, size at least $(jy-1) * m_y + iy + (n-1) * \text{abs}(incy)$. This array contains the entries of the distributed vector $\text{sub}(y)$.
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . <i>incy</i> must not be zero.
<i>a</i>	(local) COMPLEX for <code>pcgeru</code> DOUBLE COMPLEX for <code>pzgeru</code> Array, size at least $(lld_a, \text{LOCq}(ja+n-1))$. Before entry this array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

*a*Overwritten by the updated distributed matrix $\text{sub}(A)$.

p?hemv

Computes a distributed matrix-vector product using a Hermitian matrix.

Syntax

Fortran:

```
call pchemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy,
           jy, descy, incy)

call pzhemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy,
           jy, descy, incy)
```

C:

```
void pchemv (char *uplo , MKL_INT *n , float *alpha , float *a , MKL_INT *ia , MKL_INT
             *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_INT
             *incx , float *beta , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy , MKL_INT
             *incy );

void pzhemv (char *uplo , MKL_INT *n , double *alpha , double *a , MKL_INT *ia ,
             MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
             *descx , MKL_INT *incx , double *beta , double *y , MKL_INT *iy , MKL_INT *jy ,
             MKL_INT *descy , MKL_INT *incy );
```

Include Files

- C: mkl_pblas.h

Description

The p?hemv routines perform a distributed matrix-vector operation defined as

```
sub(y) := alpha*sub(A)*sub(x) + beta*sub(y),
```

where:

alpha and *beta* are scalars,

sub(A) is a *n*-by-*n* Hermitian distributed matrix, $\text{sub}(A)=A(i_a:i_a+n-1, j_a:j_a+n-1)$,

sub(x) and *sub(y)* are distributed vectors.

sub(x) denotes $X(ix, j_x:j_x+n-1)$ if *incx* = *m_x*, and $X(ix: i_x+n-1, j_x)$ if *incx* = 1,

sub(y) denotes $Y(i_y, j_y:j_y+n-1)$ if *incy* = *m_y*, and $Y(i_y: i_y+n-1, j_y)$ if *incy* = 1.

Input Parameters

uplo

(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix $\text{sub}(A)$ is used:

If *uplo* = 'U' or 'u', then the upper triangular part of the $\text{sub}(A)$ is used.

If *uplo* = 'L' or 'l', then the low triangular part of the $\text{sub}(A)$ is used.

<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>alpha</i>	(global) COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Array, size $(\text{lld}_a, \text{LOCq}(ja+n-1))$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$. Before entry when <i>uplo</i> = 'U' or 'u', the n -by- n upper triangular part of the distributed matrix $\text{sub}(A)$ must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of $\text{sub}(A)$ is not referenced, and when <i>uplo</i> = 'L' or 'l', the n -by- n lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>descA</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
<i>x</i>	(local) COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Array, size at least $(jx-1)*m_x + ix+(n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector $\text{sub}(x)$.
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.
<i>descX</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>beta</i>	(global) COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then $\text{sub}(y)$ need not be set on input.
<i>y</i>	(local) COMPLEX for pchemv DOUBLE COMPLEX for pzhemv Array, size at least $(jy-1)*m_y + iy+(n-1)*\text{abs}(incy)$.

This array contains the entries of the distributed vector $\text{sub}(y)$.

i_y, j_y	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.
desc_y	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
incy	(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . incy must not be zero.

Output Parameters

Y	Overwritten by the updated distributed vector $\text{sub}(y)$.
-----	---

p?ahemv

Computes a distributed matrix-vector product using absolute values for a Hermitian matrix.

Syntax

Fortran:

```
call pcahemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy,
            jy, descy, incy)

call pzahemv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy,
            jy, descy, incy)
```

C:

```
void pcahemv (char *uplo , MKL_INT *n , float *alpha , float *a , MKL_INT *ia ,
               MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
               MKL_INT *incx , float *beta , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy ,
               MKL_INT *incy );

void pzahemv (char *uplo , MKL_INT *n , double *alpha , double *a , MKL_INT *ia ,
               MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
               *descx , MKL_INT *incx , double *beta , double *y , MKL_INT *iy , MKL_INT *jy ,
               MKL_INT *descy , MKL_INT *incy );
```

Include Files

- C: mkl_pblas.h

Description

The p?ahemv routines perform a distributed matrix-vector operation defined as

```
sub(y) := abs(alpha)*abs(sub(A))*abs(sub(x)) + abs(beta*sub(y)),
```

where:

α and β are scalars,

$\text{sub}(A)$ is a n -by- n Hermitian distributed matrix, $\text{sub}(A)=A(ia:ia+n-1, ja:ja+n-1)$,

$\text{sub}(x)$ and $\text{sub}(y)$ are distributed vectors.

$\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $\text{inc}_x = m_x$, and $X(ix: ix+n-1, jx)$ if $\text{inc}_x = 1$,

`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if $incy = m_y$, and $Y(iy: iy+n-1, jy)$ if $incy = 1$.

Input Parameters

<code>uplo</code>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <code>sub(A)</code> is used: If <code>uplo</code> = 'U' or 'u', then the upper triangular part of the <code>sub(A)</code> is used. If <code>uplo</code> = 'L' or 'l', then the low triangular part of the <code>sub(A)</code> is used.
<code>n</code>	(global) INTEGER. Specifies the order of the distributed matrix <code>sub(A)</code> , $n \geq 0$.
<code>alpha</code>	(global) COMPLEX for <code>pcahemv</code> DOUBLE COMPLEX for <code>pzahemv</code> Specifies the scalar <code>alpha</code> .
<code>a</code>	(local) COMPLEX for <code>pcahemv</code> DOUBLE COMPLEX for <code>pzahemv</code> Array, size $(lld_a, \text{LOCq}(ja+n-1))$. This array contains the local pieces of the distributed matrix <code>sub(A)</code> . Before entry when <code>uplo</code> = 'U' or 'u', the n -by- n upper triangular part of the distributed matrix <code>sub(A)</code> must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of <code>sub(A)</code> is not referenced, and when <code>uplo</code> = 'L' or 'l', the n -by- n lower triangular part of the distributed matrix <code>sub(A)</code> must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of <code>sub(A)</code> is not referenced.
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the distributed matrix <code>A</code> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<code>desca</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <code>A</code> .
<code>x</code>	(local) COMPLEX for <code>pcahemv</code> DOUBLE COMPLEX for <code>pzahemv</code> Array, size at least $(jx-1)*m_x + ix+(n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<code>ix, jx</code>	(global) INTEGER. The row and column indices in the distributed matrix <code>X</code> indicating the first row and the first column of the submatrix <code>sub(x)</code> , respectively.
<code>descx</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <code>X</code> .
<code>incx</code>	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . <code>incx</code> must not be zero.

<i>beta</i>	(global) COMPLEX for <code>pcahemv</code> DOUBLE COMPLEX for <code>pzahemv</code> Specifies the scalar <i>beta</i> . When <i>beta</i> is set to zero, then <code>sub(y)</code> need not be set on input.
<i>y</i>	(local) COMPLEX for <code>pcahemv</code> DOUBLE COMPLEX for <code>pzahemv</code> Array, size at least $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector <code>sub(y)</code> .
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <code>sub(y)</code> , respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of <code>sub(y)</code> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.

Output Parameters

<i>y</i>	Overwritten by the updated distributed vector <code>sub(y)</code> .
----------	---

p?her

Performs a rank-1 update of a distributed Hermitian matrix.

Syntax

Fortran:

```
call pcher(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
call pzher(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
```

C:

```
void pcher (char *uplo , MKL_INT *n , float *alpha , float *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , MKL_INT *incx , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca );
void pzher (char *uplo , MKL_INT *n , double *alpha , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca );
```

Include Files

- C: `mkl_pblas.h`

Description

The `p?her` routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*conjg(sub(x)') + sub(A),
```

where:

alpha is a real scalar,

sub(A) is a n -by- n distributed Hermitian matrix, $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$,

sub(x) is distributed vector.

sub(x) denotes $X(ix, jx:jx+n-1)$ if $\text{incx} = m_x$, and $X(ix: ix+n-1, jx)$ if $\text{incx} = 1$.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(A)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(A)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(A)</i> is used.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(A)</i> , $n \geq 0$.
<i>alpha</i>	(global) REAL for pcher DOUBLE REAL for pzher Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) COMPLEX for pcher DOUBLE COMPLEX for pzher Array, size at least $(jx-1)*m_x + ix+(n-1)*\text{abs}(incx)$. This array contains the entries of the distributed vector <i>sub(x)</i> .
<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <i>sub(x)</i> , respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(x)</i> . Only two values are supported, namely 1 and m_x . <i>incx</i> must not be zero.
<i>a</i>	(local) COMPLEX for pcher DOUBLE COMPLEX for pzher Array, size $(1ld_a, \text{LOCq}(ja+n-1))$. This array contains the local pieces of the distributed matrix <i>sub(A)</i> . Before entry with <i>uplo</i> = 'U' or 'u', the n -by- n upper triangular part of the distributed matrix <i>sub(A)</i> must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of <i>sub(A)</i> is not referenced, and with <i>uplo</i> = 'L' or 'l', the n -by- n lower triangular part of the distributed matrix <i>sub(A)</i> must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of <i>sub(A)</i> is not referenced.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

<i>a</i>	With $\text{uplo} = 'U'$ or ' <u>u</u> ', the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated distributed matrix $\text{sub}(A)$. With $\text{uplo} = 'L'$ or ' <l>', the lower triangular part of the array <i>a</i> is overwritten by the lower triangular part of the updated distributed matrix $\text{sub}(A)$.</l>
----------	--

p?her2

Performs a rank-2 update of a distributed Hermitian matrix.

Syntax

Fortran:

```
call pcher2(uplo, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia,
ja, desca)
call pzher2(uplo, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia,
ja, desca)
```

C:

```
void pcher2 (char *uplo , MKL_INT *n , float *alpha , float *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT
*descy , MKL_INT *incy , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pzher2 (char *uplo , MKL_INT *n , double *alpha , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , MKL_INT *incy , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca );
```

Include Files

- C: mkl_pblas.h

Description

The p?her2 routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*conj(sub(y)') + conj(alpha)*sub(y)*conj(sub(x)') + sub(A),
```

where:

alpha is a scalar,

sub(*A*) is a *n*-by-*n* distributed Hermitian matrix, sub(*A*)=*A*(*ia*:*ia+n-1*, *ja*:*ja+n-1*),

sub(*x*) and sub(*y*) are distributed vectors.

sub(*x*) denotes *X*(*ix*, *jx*:*jx+n-1*) if *incx* = *m_x*, and *X*(*ix*: *ix+n-1*, *jx*) if *incx* = 1,

`sub(y)` denotes $Y(iy, jy:jy+n-1)$ if $incy = m_y$, and $Y(iy: iy+n-1, jy)$ if $incy = 1$.

Input Parameters

<code>uplo</code>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the distributed Hermitian matrix <code>sub(A)</code> is used: If <code>uplo</code> = 'U' or 'u', then the upper triangular part of the <code>sub(A)</code> is used. If <code>uplo</code> = 'L' or 'l', then the low triangular part of the <code>sub(A)</code> is used.
<code>n</code>	(global) INTEGER. Specifies the order of the distributed matrix <code>sub(A)</code> , $n \geq 0$.
<code>alpha</code>	(global) COMPLEX for <code>pcher2</code> DOUBLE COMPLEX for <code>pzher2</code> Specifies the scalar <code>alpha</code> .
<code>x</code>	(local) COMPLEX for <code>pcher2</code> DOUBLE COMPLEX for <code>pzher2</code> Array, size at least $(jx-1)*m_x + ix+(n-1)*abs(incx)$. This array contains the entries of the distributed vector <code>sub(x)</code> .
<code>ix, jx</code>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix <code>sub(x)</code> , respectively.
<code>descx</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<code>incx</code>	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . <code>incx</code> must not be zero.
<code>y</code>	(local) COMPLEX for <code>pcher2</code> DOUBLE COMPLEX for <code>pzher2</code> Array, size at least $(jy-1)*m_y + iy+(n-1)*abs(incy)$. This array contains the entries of the distributed vector <code>sub(y)</code> .
<code>iy, jy</code>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix <code>sub(y)</code> , respectively.
<code>descy</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
<code>incy</code>	(global) INTEGER. Specifies the increment for the elements of <code>sub(y)</code> . Only two values are supported, namely 1 and m_y . <code>incy</code> must not be zero.
<code>a</code>	(local) COMPLEX for <code>pcher2</code> DOUBLE COMPLEX for <code>pzher2</code>

Array, size (lld_a , $\text{LOCq}(ja+n-1)$). This array contains the local pieces of the distributed matrix $\text{sub}(A)$.

Before entry with $uplo = 'U'$ or ' u ', the n -by- n upper triangular part of the distributed matrix $\text{sub}(A)$ must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of $\text{sub}(A)$ is not referenced, and with $uplo = 'L'$ or ' l ', the n -by- n lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

ia, ja

(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

$desc_a$

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

a

With $uplo = 'U'$ or ' u ', the upper triangular part of the array a is overwritten by the upper triangular part of the updated distributed matrix $\text{sub}(A)$.

With $uplo = 'L'$ or ' l ', the lower triangular part of the array a is overwritten by the lower triangular part of the updated distributed matrix $\text{sub}(A)$.

p?symv

Computes a distributed matrix-vector product using a symmetric matrix.

Syntax

Fortran:

```
call pssymv(uplo, n, alpha, a, ia, ja, desc_a, x, ix, jx, desc_x, inc_x, beta, y, iy, jy, desc_y, inc_y)
```

```
call pdsymv(uplo, n, alpha, a, ia, ja, desc_a, x, ix, jx, desc_x, inc_x, beta, y, iy, jy, desc_y, inc_y)
```

C:

```
void pssymv (char *uplo , MKL_INT *n , float *alpha , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc_a , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *desc_x , MKL_INT *inc_x , float *beta , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *desc_y , MKL_INT *inc_y );
```

```
void pdsymv (char *uplo , MKL_INT *n , double *alpha , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc_a , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *desc_x , MKL_INT *inc_x , double *beta , double *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *desc_y , MKL_INT *inc_y );
```

Include Files

- C: mkl_pblas.h

Description

The p?symv routines perform a distributed matrix-vector operation defined as

```
sub(y) := alpha*sub(A)*sub(x) + beta*sub(y),
```

where:

alpha and *beta* are scalars,

sub(A) is a n -by- n symmetric distributed matrix, $\text{sub}(A)=A(ia:ia+n-1, ja:ja+n-1)$,

sub(x) and *sub(y)* are distributed vectors.

sub(x) denotes $X(ix, jx:jx+n-1)$ if $\text{incx} = m_x$, and $X(ix: ix+n-1, jx)$ if $\text{incx} = 1$,

sub(y) denotes $Y(iy, jy:jy+n-1)$ if $\text{incy} = m_y$, and $Y(iy: iy+n-1, jy)$ if $\text{incy} = 1$.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix <i>sub(A)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(A)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(A)</i> is used.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(A)</i> , $n \geq 0$.
<i>alpha</i>	(global) REAL for pssymv DOUBLE REAL for pdsymv Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) REAL for pssymv DOUBLE REAL for pdsymv Array, size $(\text{lld}_a, \text{LOCq}(ja+n-1))$. This array contains the local pieces of the distributed matrix <i>sub(A)</i> . Before entry when <i>uplo</i> = 'U' or 'u', the n -by- n upper triangular part of the distributed matrix <i>sub(A)</i> must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of <i>sub(A)</i> is not referenced, and when <i>uplo</i> = 'L' or 'l', the n -by- n lower triangular part of the distributed matrix <i>sub(A)</i> must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of <i>sub(A)</i> is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>x</i>	(local) REAL for pssymv DOUBLE REAL for pdsymv Array, size at least $(jx-1)*m_x + ix+(n-1)*\text{abs}(incx)$.

This array contains the entries of the distributed vector `sub(x)`.

<code>ix, jx</code>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix <code>sub(x)</code> , respectively.
<code>descx</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<code>incx</code>	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . <code>incx</code> must not be zero.
<code>beta</code>	(global) REAL for <code>pssymv</code> DOUBLE REAL for <code>pdsymv</code> Specifies the scalar <code>beta</code> . When <code>beta</code> is set to zero, then <code>sub(y)</code> need not be set on input.
<code>y</code>	(local) REAL for <code>pssymv</code> DOUBLE REAL for <code>pdsymv</code> Array, size at least $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector <code>sub(y)</code> .
<code>iy, jy</code>	(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix <code>sub(y)</code> , respectively.
<code>descy</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .
<code>incy</code>	(global) INTEGER. Specifies the increment for the elements of <code>sub(y)</code> . Only two values are supported, namely 1 and m_y . <code>incy</code> must not be zero.

Output Parameters

<code>y</code>	Overwritten by the updated distributed vector <code>sub(y)</code> .
----------------	---

p?asymv

Computes a distributed matrix-vector product using absolute values for a symmetric matrix.

Syntax

Fortran:

```
call psasymv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy,
            jy, descy, incy)

call pdasymv(uplo, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx, beta, y, iy,
            jy, descy, incy)
```

C:

```

void psasymv (char *uplo , MKL_INT *n , float *alpha , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx ,
MKL_INT *incx , float *beta , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT *descy ,
MKL_INT *incy );

void pdasymv (char *uplo , MKL_INT *n , double *alpha , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx , double *beta , double *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , MKL_INT *incy );

```

Include Files

- C: mkl_pblas.h

Description

The p?symv routines perform a distributed matrix-vector operation defined as

```
sub(y) := abs(alpha)*abs(sub(A))*abs(sub(x)) + abs(beta*sub(y)),
```

where:

alpha and *beta* are scalars,

sub(A) is a *n*-by-*n* symmetric distributed matrix, *sub(A)=A(ia:ia+n-1, ja:ja+n-1)*,

sub(x) and *sub(y)* are distributed vectors.

sub(x) denotes $X(ix, jx:jx+n-1)$ if *incx = m_x*, and $X(ix: ix+n-1, jx)$ if *incx = 1*,

sub(y) denotes $Y(iy, jy:jy+n-1)$ if *incy = m_y*, and $Y(iy: iy+n-1, jy)$ if *incy = 1*.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix <i>sub(A)</i> is used: If <i>uplo = 'U'</i> or <i>'u'</i> , then the upper triangular part of the <i>sub(A)</i> is used. If <i>uplo = 'L'</i> or <i>'l'</i> , then the low triangular part of the <i>sub(A)</i> is used.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(A)</i> , <i>n</i> ≥ 0 .
<i>alpha</i>	(global) REAL for psasymv DOUBLE REAL for pdasymv Specifies the scalar <i>alpha</i> .
<i>a</i>	(local)REAL for psasymv DOUBLE REAL for pdasymv Array, size (<i>lld_a, LOCq(ja+n-1)</i>). This array contains the local pieces of the distributed matrix <i>sub(A)</i> . Before entry when <i>uplo = 'U'</i> or <i>'u'</i> , the <i>n</i> -by- <i>n</i> upper triangular part of the distributed matrix <i>sub(A)</i> must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of <i>sub(A)</i> is not referenced, and when <i>uplo = 'L'</i> or <i>'l'</i> , the <i>n</i> -by- <i>n</i> lower

triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

ia, ja

(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

descA

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

x

(local) REAL for psasymv

DOUBLE PRECISION for pdasymv

Array, size at least $(jx-1)*m_x + ix + (n-1)*\text{abs}(incx)$.

This array contains the entries of the distributed vector $\text{sub}(x)$.

ix, jx

(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.

descX

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .

incx

(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

beta

(global) REAL for psasymv

DOUBLE PRECISION for pdasymv

Specifies the scalar β . When β is set to zero, then $\text{sub}(y)$ need not be set on input.

y

(local) REAL for psasymv

DOUBLE PRECISION for pdasymv

Array, size at least $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$.

This array contains the entries of the distributed vector $\text{sub}(y)$.

iy, jy

(global) INTEGER. The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.

descY

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix Y .

incy

(global) INTEGER. Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and m_y . $incy$ must not be zero.

Output Parameters

y

Overwritten by the updated distributed vector $\text{sub}(y)$.

p?syr

Performs a rank-1 update of a distributed symmetric matrix.

Syntax**Fortran:**

```
call pssyr(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
call pdssyr(uplo, n, alpha, x, ix, jx, descx, incx, a, ia, ja, desca)
```

C:

```
void pssyr (char *uplo , MKL_INT *n , float *alpha , float *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , MKL_INT *incx , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca );
void pdssyr (char *uplo , MKL_INT *n , double *alpha , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , double *a , MKL_INT *ia , MKL_INT *ja ,
MKL_INT *desca );
```

Include Files

- C: mkl_pblas.h

Description

The p?syr routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*sub(x)' + sub(A),
```

where:

alpha is a scalar,

sub(A) is a *n*-by-*n* distributed symmetric matrix, *sub(A)=A(ia:ia+n-1, ja:ja+n-1)*,

sub(x) is distributed vector.

sub(x) denotes *X(ix, jx:jx+n-1)* if *incx = m_x*, and *X(ix: ix+n-1, jx)* if *incx = 1*,

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix <i>sub(A)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(A)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(A)</i> is used.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(A)</i> , <i>n</i> ≥ 0.
<i>alpha</i>	(global) REAL for pssyr DOUBLE REAL for pdssyr Specifies the scalar <i>alpha</i> .
<i>x</i>	(local)REAL for pssyr DOUBLE REAL for pdssyr

Array, size at least $(jx-1) * m_x + ix + (n-1) * \text{abs}(incx)$.

This array contains the entries of the distributed vector $\text{sub}(x)$.

ix, jx

(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.

$descx$

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .

$incx$

(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

a

(local) REAL for pssyr

DOUBLE REAL for pdssyr

Array, size $(lld_a, \text{LOCq}(ja+n-1))$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$.

Before entry with $uplo = 'U'$ or ' u ', the n -by- n upper triangular part of the distributed matrix $\text{sub}(A)$ must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of $\text{sub}(A)$ is not referenced, and with $uplo = 'L'$ or ' l ', the n -by- n lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

ia, ja

(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

$desca$

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

Output Parameters

a

With $uplo = 'U'$ or ' u ', the upper triangular part of the array a is overwritten by the upper triangular part of the updated distributed matrix $\text{sub}(A)$.

With $uplo = 'L'$ or ' l ', the lower triangular part of the array a is overwritten by the lower triangular part of the updated distributed matrix $\text{sub}(A)$.

p?syr2

Performs a rank-2 update of a distributed symmetric matrix.

Syntax

Fortran:

```
call pssyr2(uplo, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia,
ja, desca)
```

```
call pdssyr2(uplo, n, alpha, x, ix, jx, descx, incx, y, iy, jy, descy, incy, a, ia,
ja, desca)
```

C:

```
void pssyr2 (char *uplo , MKL_INT *n , float *alpha , float *x , MKL_INT *ix , MKL_INT
*jx , MKL_INT *descx , MKL_INT *incx , float *y , MKL_INT *iy , MKL_INT *jy , MKL_INT
*descy , MKL_INT *incy , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pdssyr2 (char *uplo , MKL_INT *n , double *alpha , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , double *y , MKL_INT *iy , MKL_INT *jy ,
MKL_INT *descy , MKL_INT *incy , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT
*desca );
```

Include Files

- C: mkl_pblas.h

Description

The p?ssyr2 routines perform a distributed matrix-vector operation defined as

```
sub(A) := alpha*sub(x)*sub(y)' + alpha*sub(y)*sub(x)' + sub(A),
```

where:

alpha is a scalar,

sub(A) is a *n*-by-*n* distributed symmetric matrix, *sub(A)*=*A*(*ia*:*ia+n-1*, *ja*:*ja+n-1*) ,

sub(x) and *sub(y)* are distributed vectors.

sub(x) denotes *X*(*ix*, *jx*:*jx+n-1*) if *incx* = *m_x*, and *X*(*ix*: *ix+n-1*, *jx*) if *incx* = 1,

sub(y) denotes *Y*(*iy*, *jy*:*jy+n-1*) if *incy* = *m_y*, and *Y*(*iy*: *iy+n-1*, *jy*) if *incy* = 1.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the distributed symmetric matrix <i>sub(A)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(A)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(A)</i> is used.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(A)</i> , <i>n</i> ≥ 0.
<i>alpha</i>	(global) REAL for pssyr2 DOUBLE REAL for pdssyr2 Specifies the scalar <i>alpha</i> .
<i>x</i>	(local) REAL for pssyr2 DOUBLE REAL for pdssyr2 Array, size at least (<i>jx-1</i> * <i>m_x</i> + <i>ix</i> +(<i>n-1</i>)*abs(<i>incx</i>)). This array contains the entries of the distributed vector <i>sub(x)</i> .

<i>ix, jx</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>X</i> indicating the first row and the first column of the submatrix <i>sub(x)</i> , respectively.
<i>descx</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>X</i> .
<i>incx</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(x)</i> . Only two values are supported, namely 1 and <i>m_x</i> . <i>incx</i> must not be zero.
<i>y</i>	(local) REAL for pssyr2 DOUBLE REAL for pdssyr2 Array, size at least $(jy-1)*m_y + iy + (n-1)*\text{abs}(incy)$. This array contains the entries of the distributed vector <i>sub(y)</i> .
<i>iy, jy</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <i>sub(y)</i> , respectively.
<i>descy</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(y)</i> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.
<i>a</i>	(local) REAL for pssyr2 DOUBLE REAL for pdssyr2 Array, size $(lld_a, \text{LOCq}(ja+n-1))$. This array contains the local pieces of the distributed matrix <i>sub(A)</i> . Before entry with <i>uplo</i> = 'U' or 'u', the <i>n</i> -by- <i>n</i> upper triangular part of the distributed matrix <i>sub(A)</i> must contain the upper triangular part of the distributed symmetric matrix and the strictly lower triangular part of <i>sub(A)</i> is not referenced, and with <i>uplo</i> = 'L' or 'l', the <i>n</i> -by- <i>n</i> lower triangular part of the distributed matrix <i>sub(A)</i> must contain the lower triangular part of the distributed symmetric matrix and the strictly upper triangular part of <i>sub(A)</i> is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .

Output Parameters

<i>a</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of the array <i>a</i> is overwritten by the upper triangular part of the updated distributed matrix <i>sub(A)</i> .
----------	--

With $uplo = 'L'$ or ' l ', the lower triangular part of the array a is overwritten by the lower triangular part of the updated distributed matrix $\text{sub}(A)$.

p?trmv

Computes a distributed matrix-vector product using a triangular matrix.

Syntax

Fortran:

```
call pstrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pdtrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pctrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pztrmv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
```

C:

```
void pstrmv (char *uplo , char *trans , char *diag , MKL_INT *n , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pdtrmv (char *uplo , char *trans , char *diag , MKL_INT *n , double *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pctrmv (char *uplo , char *trans , char *diag , MKL_INT *n , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pztrmv (char *uplo , char *trans , char *diag , MKL_INT *n , double *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
```

Include Files

- C: mkl_pblas.h

Description

The p?trmv routines perform one of the following distributed matrix-vector operations defined as

$\text{sub}(x) := \text{sub}(A) * \text{sub}(x)$, or $\text{sub}(x) := \text{sub}(A)' * \text{sub}(x)$, or $\text{sub}(x) := \text{conjg}(\text{sub}(A)') * \text{sub}(x)$,
where:

$\text{sub}(A)$ is a n -by- n unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$,

$\text{sub}(x)$ is an n -element distributed vector.

$\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $incx = m_x$, and $X(ix: ix+n-1, jx)$ if $incx = 1$,

Input Parameters

uplo (global) CHARACTER*1. Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular:

if *uplo* = 'U' or 'u', then the matrix is upper triangular;
 if *uplo* = 'L' or 'l', then the matrix is low triangular.

trans

(global) CHARACTER*1. Specifies the form of *op*(*sub*(*A*)) used in the matrix equation:

if *transa* = 'N' or 'n', then *sub*(*x*) := *sub*(*A*) * *sub*(*x*);
 if *transa* = 'T' or 't', then *sub*(*x*) := *sub*(*A*)' * *sub*(*x*);
 if *transa* = 'C' or 'c', then *sub*(*x*) := *conjg*(*sub*(*A*)') * *sub*(*x*).

diag

(global) CHARACTER*1. Specifies whether the matrix *sub*(*A*) is unit triangular:

if *diag* = 'U' or 'u' then the matrix is unit triangular;
 if *diag* = 'N' or 'n', then the matrix is not unit triangular.

n

(global) INTEGER. Specifies the order of the distributed matrix *sub*(*A*), *n*≥0.

a

(local)REAL for *pstrmv*

DOUBLE PRECISION for *pdtrmv*

COMPLEX for *pctrmv*

DOUBLE COMPLEX for *pztrmv*

Array, size at least (*lld_a*, *LOCq*(1, *ja*+*n*-1)).

Before entry with *uplo* = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix *sub*(*A*), and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix *sub*(*A*) is not referenced.

Before entry with *uplo* = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix *sub*(*A*), and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix *sub*(*A*) is not referenced .

When *diag* = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix *sub*(*A*) are not referenced either, but are assumed to be unity.

ia, ja

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix *sub*(*A*), respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

x

(local)REAL for *pstrmv*

DOUBLE PRECISION for *pdtrmv*

COMPLEX for *pctrmv*

DOUBLE COMPLEX for *pztrmv*

Array, size at least (*jx*-1)**m_x* + *ix*+(*n*-1)*abs(*incx*)).

This array contains the entries of the distributed vector `sub(x)`.

<code>ix, jx</code>	(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix <code>sub(x)</code> , respectively.
<code>descx</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .
<code>incx</code>	(global) INTEGER. Specifies the increment for the elements of <code>sub(x)</code> . Only two values are supported, namely 1 and m_x . <code>incx</code> must not be zero.

Output Parameters

<code>x</code>	Overwritten by the transformed distributed vector <code>sub(x)</code> .
----------------	---

p?atrmv

Computes a distributed matrix-vector product using absolute values for a triangular matrix.

Syntax

Fortran:

```
call psatrmv(uplo, trans, diag, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx,
beta, y, iy, jy, descy, incy)

call pdatrmv(uplo, trans, diag, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx,
beta, y, iy, jy, descy, incy)

call pcatrmv(uplo, trans, diag, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx,
beta, y, iy, jy, descy, incy)

call pzatrmv(uplo, trans, diag, n, alpha, a, ia, ja, desca, x, ix, jx, descx, incx,
beta, y, iy, jy, descy, incy)
```

C:

```
void psatrmv (char *uplo , char *trans , char *diag , MKL_INT *n , float *alpha ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , float *beta , float *y , MKL_INT *iy ,
MKL_INT *jy , MKL_INT *descy , MKL_INT *incy );

void pdatrmv (char *uplo , char *trans , char *diag , MKL_INT *n , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , double *beta , double *y , MKL_INT *iy ,
MKL_INT *jy , MKL_INT *descy , MKL_INT *incy );

void pcatrmv (char *uplo , char *trans , char *diag , MKL_INT *n , float *alpha ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , float *beta , float *y , MKL_INT *iy ,
MKL_INT *jy , MKL_INT *descy , MKL_INT *incy );

void pzatrmv (char *uplo , char *trans , char *diag , MKL_INT *n , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix ,
MKL_INT *jx , MKL_INT *descx , MKL_INT *incx , double *beta , double *y , MKL_INT *iy ,
MKL_INT *jy , MKL_INT *descy , MKL_INT *incy );
```

Include Files

- C: mkl_pblas.h

Description

The p?atrmv routines perform one of the following distributed matrix-vector operations defined as

```
sub(y) := abs(alpha)*abs(sub(A))*abs(sub(x)) + abs(beta*sub(y)), or
sub(y) := abs(alpha)*abs(sub(A'))*abs(sub(x)) + abs(beta*sub(y)), or
sub(y) := abs(alpha)*abs(conjg(sub(A')))*abs(sub(x)) + abs(beta*sub(y)),
```

where:

alpha and *beta* are scalars,

sub(A) is a *n*-by-*n* unit, or non-unit, upper or lower triangular distributed matrix, *sub(A)* = *A*(ia:ia+n-1, ja:ja+n-1),

sub(x) is an *n*-element distributed vector.

sub(x) denotes *X(ix, jx:jx+n-1)* if *incx* = *m_x*, and *X(ix: ix+n-1, jx)* if *incx* = 1.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the distributed matrix <i>sub(A)</i> is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	(global) CHARACTER*1. Specifies the form of <i>op</i> (<i>sub(A)</i>) used in the matrix equation: if <i>trans</i> = 'N' or 'n', then <i>sub(y)</i> := <i>alpha</i> * <i>sub(A)</i> * <i>sub(x)</i> + <i>beta</i> * <i>sub(y)</i> ; if <i>trans</i> = 'T' or 't', then <i>sub(y)</i> := <i>alpha</i> * <i>sub(A)</i> ' * <i>sub(x)</i> + <i>beta</i> * <i>sub(y)</i> ; if <i>trans</i> = 'C' or 'c', then <i>sub(y)</i> := <i>alpha</i> * <i>conjg</i> (<i>sub(A)</i> ') * <i>sub(x)</i> + <i>beta</i> * <i>sub(y)</i> .
<i>diag</i>	(global) CHARACTER*1. Specifies whether the matrix <i>sub(A)</i> is unit triangular: if <i>diag</i> = 'U' or 'u' then the matrix is unit triangular; if <i>diag</i> = 'N' or 'n', then the matrix is not unit triangular.
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(A)</i> , <i>n</i> ≥0.
<i>alpha</i>	(global) REAL for psatrmv DOUBLE PRECISION for pdatrmv COMPLEX for pcatrmv DOUBLE COMPLEX for pzatrmv Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) REAL for psatrmv

DOUBLE PRECISION for pdatrmv

COMPLEX for pcatrmv

DOUBLE COMPLEX for pzatrmv

Array, size at least ($l1d_a$, LOCq(1, $ja+n-1$)).

Before entry with $uplo = 'U'$ or ' u ', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.

Before entry with $uplo = 'L'$ or ' l ', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.

When $diag = 'U'$ or ' u ', the local entries corresponding to the diagonal elements of the submatrix $\text{sub}(A)$ are not referenced either, but are assumed to be unity.

ia, ja

(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

$desc_a$

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

x

(local)REAL for psatrmv

DOUBLE PRECISION for pdatrmv

COMPLEX for pcatrmv

DOUBLE COMPLEX for pzatrmv

Array, size at least ($jx-1) * m_x + ix + (n-1) * \text{abs}(incx)$).

This array contains the entries of the distributed vector $\text{sub}(x)$.

ix, jx

(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.

$desc_x$

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .

$incx$

(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

$beta$

(global)REAL for psatrmv

DOUBLE PRECISION for pdatrmv

COMPLEX for pcatrmv

DOUBLE COMPLEX for pzatrmv

Specifies the scalar β . When β is set to zero, then $\text{sub}(y)$ need not be set on input.

<i>y</i>	(local)REAL for psatrmv DOUBLE PRECISION for pdatrmv COMPLEX for pcatrmv DOUBLE COMPLEX for pzatrmv Array, size $(jy-1)*m_y + iy + (m-1)*abs(incy)$ when <i>trans</i> = 'N' or 'n', and $(jy-1)*m_y + iy + (n-1)*abs(incy)$ otherwise. This array contains the entries of the distributed vector <i>sub(y)</i> .
<i>i_y, j_y</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>Y</i> indicating the first row and the first column of the submatrix <i>sub(y)</i> , respectively.
<i>desc_y</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>Y</i> .
<i>incy</i>	(global) INTEGER. Specifies the increment for the elements of <i>sub(y)</i> . Only two values are supported, namely 1 and <i>m_y</i> . <i>incy</i> must not be zero.

Output Parameters

<i>x</i>	Overwritten by the transformed distributed vector <i>sub(y)</i> .
----------	---

p?trsv

Solves a system of linear equations whose coefficients are in a distributed triangular matrix.

Syntax

Fortran:

```
call pstrsv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pdtrsv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pctrsrv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
call pztrsv(uplo, trans, diag, n, a, ia, ja, desca, x, ix, jx, descx, incx)
```

C:

```
void pstrsv (char *uplo , char *trans , char *diag , MKL_INT *n , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pdtrsv (char *uplo , char *trans , char *diag , MKL_INT *n , double *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pctrsrv (char *uplo , char *trans , char *diag , MKL_INT *n , float *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
void pztrsv (char *uplo , char *trans , char *diag , MKL_INT *n , double *a , MKL_INT
*ia , MKL_INT *ja , MKL_INT *desca , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT
*descx , MKL_INT *incx );
```

Include Files

- C: mkl_pblas.h

Description

The p?trsv routines solve one of the systems of equations:

$\text{sub}(A) * \text{sub}(x) = b$, or $\text{sub}(A)' * \text{sub}(x) = b$, or $\text{conjg}(\text{sub}(A)') * \text{sub}(x) = b$,

where:

$\text{sub}(A)$ is a n -by- n unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A) = A(i_a:i_a+n-1, j_a:j_a+n-1)$,

b and $\text{sub}(x)$ are n -element distributed vectors,

$\text{sub}(x)$ denotes $X(ix, jx:jx+n-1)$ if $\text{incx} = m_x$, and $X(ix: ix+n-1, jx)$ if $\text{incx} = 1$.

The routine does not test for singularity or near-singularity. Such tests must be performed before calling this routine.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular: if $\text{uplo} = 'U'$ or ' <u>u</u> ', then the matrix is upper triangular; if $\text{uplo} = 'L'$ or ' <l>', then the matrix is low triangular.</l>
<i>trans</i>	(global) CHARACTER*1. Specifies the form of the system of equations: if $\text{transa} = 'N'$ or ' <n>', then $\text{sub}(A) * \text{sub}(x) = b$; if $\text{transa} = 'T'$ or '<t>', then $\text{sub}(A)' * \text{sub}(x) = b$; if $\text{transa} = 'C'$ or '<c>', then $\text{conjg}(\text{sub}(A)') * \text{sub}(x) = b$.</c></t></n>
<i>diag</i>	(global) CHARACTER*1. Specifies whether the matrix $\text{sub}(A)$ is unit triangular: if $\text{diag} = 'U'$ or ' <u>u</u> ' then the matrix is unit triangular; if $\text{diag} = 'N'$ or ' <n>', then the matrix is not unit triangular.</n>
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.
<i>a</i>	(local) REAL for pstrsv DOUBLE PRECISION for pdtrsv COMPLEX for pctrs DOUBLE COMPLEX for pztrsv Array, size at least $(\text{lld}_a, \text{LOCq}(1, j_a+n-1))$. Before entry with $\text{uplo} = 'U'$ or ' <u>u</u> ', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.

Before entry with $\text{uplo} = \text{'L'}$ or 'l' , this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $\text{sub}(A)$ is not referenced .

When $\text{diag} = \text{'U'}$ or 'u' , the local entries corresponding to the diagonal elements of the submatrix $\text{sub}(A)$ are not referenced either, but are assumed to be unity.

ia, ja

(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

$desc_a$

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

x

(local)REAL for pstrsv

DOUBLE PRECISION for pdtrsv

COMPLEX for pcptrsv

DOUBLE COMPLEX for pzptrsv

Array, size at least $(jx-1)*m_x + ix + (n-1)*\text{abs}(incx)$.

This array contains the entries of the distributed vector $\text{sub}(x)$. Before entry, $\text{sub}(x)$ must contain the n -element right-hand side distributed vector b .

ix, jx

(global) INTEGER. The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.

$desc_x$

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix X .

$incx$

(global) INTEGER. Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and m_x . $incx$ must not be zero.

Output Parameters

x

Overwritten with the solution vector.

PBLAS Level 3 Routines

The PBLAS Level 3 routines perform distributed matrix-matrix operations. [Table "PBLAS Level 3 Routine Groups and Their Data Types"](#) lists the PBLAS Level 3 routine groups and the data types associated with them.

PBLAS Level 3 Routine Groups and Their Data Types

Routine Group	Data Types	Description
<code>p?geadd</code>	s, d, c, z	Distributed matrix-matrix sum of general matrices
<code>p?tradd</code>	s, d, c, z	Distributed matrix-matrix sum of triangular matrices

Routine Group	Data Types	Description
p?gemm	s, d, c, z	Distributed matrix-matrix product of general matrices
p?hemm	c, z	Distributed matrix-matrix product, one matrix is Hermitian
p?herk	c, z	Rank-k update of a distributed Hermitian matrix
p?her2k	c, z	Rank-2k update of a distributed Hermitian matrix
p?somm	s, d, c, z	Matrix-matrix product of distributed symmetric matrices
p?syrk	s, d, c, z	Rank-k update of a distributed symmetric matrix
p?syr2k	s, d, c, z	Rank-2k update of a distributed symmetric matrix
p?tran	s, d	Transposition of a real distributed matrix
p?tranc	c, z	Transposition of a complex distributed matrix (conjugated)
p?tranu	c, z	Transposition of a complex distributed matrix
p?trmm	s, d, c, z	Distributed matrix-matrix product, one matrix is triangular
p?trsm	s, d, c, z	Solution of a distributed matrix equation, one matrix is triangular

p?geadd

Performs sum operation for two distributed general matrices.

Syntax

Fortran:

```
call psgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pdgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pcgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
call pzgeadd(trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descc)
```

C:

```
void psgeadd (char *trans , MKL_INT *m , MKL_INT *n , float *alpha , float *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *beta , float *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc );
void pdgeadd (char *trans , MKL_INT *m , MKL_INT *n , double *alpha , double *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *beta , double *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc );
void pcgeadd (char *trans , MKL_INT *m , MKL_INT *n , float *alpha , float *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *beta , float *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc );
void pzgeadd (char *trans , MKL_INT *m , MKL_INT *n , double *alpha , double *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *beta , double *c , MKL_INT *ic ,
MKL_INT *jc , MKL_INT *descc );
```

Include Files

- C: mkl_pblas.h

Description

The p?geadd routines perform sum operation for two distributed general matrices. The operation is defined as

```
sub(C) := beta*sub(C) + alpha*op(sub(A)),
```

where:

op(x) is one of *op(x) = x*, or *op(x) = x'*,

alpha and *beta* are scalars,

sub(C) is an *m*-by-*n* distributed matrix, *sub(C)=C(ic:ic+m-1, jc:jc+n-1)*.

sub(A) is a distributed matrix, *sub(A)=A(ia:ia+n-1, ja:ja+m-1)*.

Input Parameters

<i>trans</i>	(global) CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then <i>op(sub(A)) := sub(A)</i> ; if <i>trans</i> = 'T' or 't', then <i>op(sub(A)) := sub(A)'</i> ; if <i>trans</i> = 'C' or 'c', then <i>op(sub(A)) := sub(A)''</i> .
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix <i>sub(C)</i> and the number of columns of the submatrix <i>sub(A)</i> , <i>m</i> ≥ 0.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix <i>sub(C)</i> and the number of rows of the submatrix <i>sub(A)</i> , <i>n</i> ≥ 0.
<i>alpha</i>	(global) REAL for psgeadd DOUBLE PRECISION for pdgeadd COMPLEX for pcgeadd DOUBLE COMPLEX for pzgeadd Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) REAL for psgeadd DOUBLE PRECISION for pdgeadd COMPLEX for pcgeadd DOUBLE COMPLEX for pzgeadd Array, size (<i>lld_a, LOCq(ja+m-1)</i>). This array contains the local pieces of the distributed matrix <i>sub(A)</i> .
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <i>sub(A)</i> , respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .

<i>beta</i>	(global)REAL for psgeadd DOUBLE PRECISION for pdgeadd COMPLEX for pcgeadd DOUBLE COMPLEX for pzgeadd Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then <i>sub(C)</i> need not be set on input.
<i>c</i>	(local)REAL for psgeadd DOUBLE PRECISION for pdgeadd COMPLEX for pcgeadd DOUBLE COMPLEX for pzgeadd Array, size (<i>lld_c</i> , LOCq(<i>jc+n-1</i>)). This array contains the local pieces of the distributed matrix <i>sub(C)</i> .
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <i>sub(C)</i> , respectively.
<i>descC</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the updated submatrix.
----------	---------------------------------------

p?tradd

Performs sum operation for two distributed triangular matrices.

Syntax

Fortran:

```
call pstradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
call pdtradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
call pctradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
call pztradd(uplo, trans, m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
```

C:

```
void pstradd (char *uplo , char *trans , MKL_INT *m , MKL_INT *n , float *alpha ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *beta , float *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descC );
void pdtradd (char *uplo , char *trans , MKL_INT *m , MKL_INT *n , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *beta , double *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descC );
```

```
void pcstradd (char *uplo , char *trans , MKL_INT *m , MKL_INT *n , float *alpha ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *beta , float *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *desc );
void pzstradd (char *uplo , char *trans , MKL_INT *m , MKL_INT *n , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *beta , double *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *desc );
```

Include Files

- C: mkl_pblas.h

Description

The p?stradd routines perform sum operation for two distributed triangular matrices. The operation is defined as

```
sub(C) := beta*sub(C) + alpha*op(sub(A)),
```

where:

op(x) is one of *op(x) = x*, or *op(x) = x'*, or *op(x) = conjg(x')*.

alpha and *beta* are scalars,

sub(C) is an *m*-by-*n* distributed matrix, *sub(C)=C(ic:ic+m-1, jc:jc+n-1)*.

sub(A) is a distributed matrix, *sub(A)=A(ia:ia+n-1, ja:ja+m-1)*.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the distributed matrix <i>sub(C)</i> is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>trans</i>	(global) CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then <i>op(sub(A)) := sub(A)</i> ; if <i>trans</i> = 'T' or 't', then <i>op(sub(A)) := sub(A)'</i> ; if <i>trans</i> = 'C' or 'c', then <i>op(sub(A)) := conjg(sub(A)')</i> .
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix <i>sub(C)</i> and the number of columns of the submatrix <i>sub(A)</i> , <i>m</i> ≥ 0.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix <i>sub(C)</i> and the number of rows of the submatrix <i>sub(A)</i> , <i>n</i> ≥ 0.
<i>alpha</i>	(global) REAL for pcstradd DOUBLE PRECISION for pdstradd COMPLEX for pcstradd DOUBLE COMPLEX for pdstradd Specifies the scalar <i>alpha</i> .
<i>a</i>	(local)REAL for pcstradd DOUBLE PRECISION for pdstradd

	COMPLEX for <code>pctradd</code> DOUBLE COMPLEX for <code>pztradd</code> Array, size (lld_a , $\text{LOCq}(ja+m-1)$). This array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
<i>beta</i>	(global) REAL for <code>pstradd</code> DOUBLE PRECISION for <code>pdtradd</code> COMPLEX for <code>pctradd</code> DOUBLE COMPLEX for <code>pztradd</code> Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then $\text{sub}(C)$ need not be set on input.
<i>c</i>	(local) REAL for <code>pstradd</code> DOUBLE PRECISION for <code>pdtradd</code> COMPLEX for <code>pctradd</code> DOUBLE COMPLEX for <code>pztradd</code> Array, size (lld_c , $\text{LOCq}(jc+n-1)$). This array contains the local pieces of the distributed matrix $\text{sub}(C)$.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.
<i>descC</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

c Overwritten by the updated submatrix.

p?gemm

Computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product for distributed matrices.

Syntax

Fortran:

```
call psgemm(transa, transb, m, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descC)
```

```
call pdgemm(transa, transb, m, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta,  

c, ic, jc, descc)  
  

call pcgemm(transa, transb, m, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta,  

c, ic, jc, descc)  
  

call pzgemm(transa, transb, m, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta,  

c, ic, jc, descc)
```

C:

```
void psgemm (char *transa , char *transb , MKL_INT *m , MKL_INT *n , MKL_INT *k ,  

float *alpha , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b ,  

MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *beta , float *c , MKL_INT *ic ,  

MKL_INT *jc , MKL_INT *descc );  
  

void pdgemm (char *transa , char *transb , MKL_INT *m , MKL_INT *n , MKL_INT *k ,  

double *alpha , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b ,  

MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *beta , double *c , MKL_INT *ic ,  

MKL_INT *jc , MKL_INT *descc );  
  

void pcgemm (char *transa , char *transb , MKL_INT *m , MKL_INT *n , MKL_INT *k ,  

float *alpha , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b ,  

MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *beta , float *c , MKL_INT *ic ,  

MKL_INT *jc , MKL_INT *descc );  
  

void pzgemm (char *transa , char *transb , MKL_INT *m , MKL_INT *n , MKL_INT *k ,  

double *alpha , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b ,  

MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *beta , double *c , MKL_INT *ic ,  

MKL_INT *jc , MKL_INT *descc );
```

Include Files

- C: mkl_pblas.h

Description

The *p?gemm* routines perform a matrix-matrix operation with general distributed matrices. The operation is defined as

```
sub(C) := alpha*op(sub(A))*op(sub(B)) + beta*sub(C),
```

where:

op(x) is one of *op(x) = x*, or *op(x) = x'*,

alpha and *beta* are scalars,

sub(A)=A(ia:ia+m-1, ja:ja+k-1), *sub(B)=B(ib:ib+k-1, jb:jb+n-1)*, and *sub(C)=C(ic:ic+m-1, jc:jc+n-1)*, are distributed matrices.

Input Parameters

<i>transa</i>	(global) CHARACTER*1. Specifies the form of <i>op(sub(A))</i> used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then <i>op(sub(A)) = sub(A)</i> ; if <i>transa</i> = 'T' or 't', then <i>op(sub(A)) = sub(A)'</i> ; if <i>transa</i> = 'C' or 'c', then <i>op(sub(A)) = sub(A)''</i> .
---------------	---

<i>transb</i>	(global) CHARACTER*1. Specifies the form of $\text{op}(\text{sub}(B))$ used in the matrix multiplication: if $\text{transb} = \text{'N'}$ or 'n' , then $\text{op}(\text{sub}(B)) = \text{sub}(B)$; if $\text{transb} = \text{'T'}$ or 't' , then $\text{op}(\text{sub}(B)) = \text{sub}(B)'$; if $\text{transb} = \text{'C'}$ or 'c' , then $\text{op}(\text{sub}(B)) = \text{sub}(B)''$.
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrices $\text{op}(\text{sub}(A))$ and $\text{sub}(C)$, $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrices $\text{op}(\text{sub}(B))$ and $\text{sub}(C)$, $n \geq 0$. The value of <i>n</i> must be at least zero.
<i>k</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix $\text{op}(\text{sub}(A))$ and the number of rows of the distributed matrix $\text{op}(\text{sub}(B))$. The value of <i>k</i> must be greater than or equal to 0.
<i>alpha</i>	(global) REAL for psgemm DOUBLE PRECISION for pdgemm COMPLEX for pcgemm DOUBLE COMPLEX for pzgemm Specifies the scalar <i>alpha</i> . When <i>alpha</i> is equal to zero, then the local entries of the arrays <i>a</i> and <i>b</i> corresponding to the entries of the submatrices $\text{sub}(A)$ and $\text{sub}(B)$ respectively need not be set on input.
<i>a</i>	(local) REAL for psgemm DOUBLE PRECISION for pdgemm COMPLEX for pcgemm DOUBLE COMPLEX for pzgemm Array, size $l1d_a$ by <i>kla</i> , where <i>kla</i> is $\text{LOCc}(ja+k-1)$ when $\text{transa} = \text{'N'}$ or 'n' , and is $\text{LOCq}(ja+m-1)$ otherwise. Before entry this array must contain the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>b</i>	(local) REAL for psgemm DOUBLE PRECISION for pdgemm COMPLEX for pcgemm DOUBLE COMPLEX for pzgemm

Array, size lld_b by klb , where klb is $\text{LOCc}(jb+n-1)$ when $\text{transb} = \text{'N'}$ or ' n ', and is $\text{LOCq}(jb+k-1)$ otherwise. Before entry this array must contain the local pieces of the distributed matrix $\text{sub}(B)$.

ib, jb	(global) INTEGER. The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively
$descb$	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix B .
β	(global) REAL for psgemm DOUBLE PRECISION for pdgemm COMPLEX for pcgemm DOUBLE COMPLEX for pzgemm Specifies the scalar β . When β is equal to zero, then $\text{sub}(C)$ need not be set on input.
c	(local) REAL for psgemm DOUBLE PRECISION for pdgemm COMPLEX for pcgemm DOUBLE COMPLEX for pzgemm Array, size $(lld_a, \text{LOCq}(jc+n-1))$. Before entry this array must contain the local pieces of the distributed matrix $\text{sub}(C)$.
ic, jc	(global) INTEGER. The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively
$descC$	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

c	Overwritten by the m -by- n distributed matrix $\alpha * \text{op}(\text{sub}(A)) * \text{op}(\text{sub}(B)) + \beta * \text{sub}(C)$.
-----	---

p?hemm

Performs a scalar-matrix-matrix product (one matrix operand is Hermitian) and adds the result to a scalar-matrix product.

Syntax

Fortran:

```
call pchemm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descC)
call pzhemm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descC)
```

C:

```
void pchemm (char *side , char *uplo , MKL_INT *m , MKL_INT *n , float *alpha , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT
*jb , MKL_INT *descb , float *beta , float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT
*descc );

void pzhemm (char *side , char *uplo , MKL_INT *m , MKL_INT *n , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , double *beta , double *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc );
```

Include Files

- C: mkl_pblas.h

Description

The p?hemm routines perform a matrix-matrix operation with distributed matrices. The operation is defined as
 $\text{sub}(C) := \alpha \text{sub}(A) * \text{sub}(B) + \beta \text{sub}(C)$,

or

$\text{sub}(C) := \alpha \text{sub}(B) * \text{sub}(A) + \beta \text{sub}(C)$,

where:

alpha and *beta* are scalars,

sub(A) is a Hermitian distributed matrix, $\text{sub}(A) = A(ia:ia+m-1, ja:ja+m-1)$, if *side* = 'L', and
 $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$, if *side* = 'R'.

sub(B) and *sub(C)* are *m*-by-*n* distributed matrices.

$\text{sub}(B) = B(ib:ib+m-1, jb:jb+n-1)$, $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$.

Input Parameters

side

(global) CHARACTER*1. Specifies whether the Hermitian distributed matrix *sub(A)* appears on the left or right in the operation:

if *side* = 'L' or 'l', then $\text{sub}(C) := \alpha \text{sub}(A) * \text{sub}(B) + \beta \text{sub}(C)$;

if *side* = 'R' or 'r', then $\text{sub}(C) := \alpha \text{sub}(B) * \text{sub}(A) + \beta \text{sub}(C)$.

uplo

(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix *sub(A)* is used:

if *uplo* = 'U' or 'u', then the upper triangular part is used;

if *uplo* = 'L' or 'l', then the lower triangular part is used.

m

(global) INTEGER. Specifies the number of rows of the distribute submatrix *sub(C)*, $m \geq 0$.

n

(global) INTEGER. Specifies the number of columns of the distribute submatrix *sub(C)*, $n \geq 0$.

alpha

(global) COMPLEX for pchemm

DOUBLE COMPLEX for pzhemm

Specifies the scalar *alpha*.

a

(local) COMPLEX for pchemm

DOUBLE COMPLEX for pzhemm

Array, size (*lld_a*, LOCq(*ja+na-1*)).

Before entry this array must contain the local pieces of the symmetric distributed matrix sub(*A*), such that when *uplo* = 'U' or 'u', the *na*-by-*na* upper triangular part of the distributed matrix sub(*A*) must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of sub(*A*) is not referenced, and when *uplo* = 'L' or 'l', the *na*-by-*na* lower triangular part of the distributed matrix sub(*A*) must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of sub(*A*) is not referenced.

ia, ja

(global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix sub(*A*), respectively

desc_a

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

b

(local) COMPLEX for pchemm

DOUBLE COMPLEX for pzhemm

Array, size (*lld_b*, LOCq(*jb+n-1*)). Before entry this array must contain the local pieces of the distributed matrix sub(*B*).

ib, jb

(global) INTEGER. The row and column indices in the distributed matrix *B* indicating the first row and the first column of the submatrix sub(*B*), respectively.

desc_b

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *B*.

beta

(global) COMPLEX for pchemm

DOUBLE COMPLEX for pzhemm

Specifies the scalar *beta*.

When *beta* is set to zero, then sub(*C*) need not be set on input.

c

(local) COMPLEX for pchemm

DOUBLE COMPLEX for pzhemm

Array, size (*lld_c*, LOCq(*jc+n-1*)). Before entry this array must contain the local pieces of the distributed matrix sub(*C*).

ic, jc

(global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix sub(*C*), respectively

desc_c

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c Overwritten by the m -by- n updated distributed matrix.

p?herk

Performs a rank- k update of a distributed Hermitian matrix.

Syntax

Fortran:

```
call pcherk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, desc)
call pzherk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, desc)
```

C:

```
void pcherk (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , float *alpha , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *beta , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *desc );
void pzherk (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *beta , double *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *desc );
```

Include Files

- C: mkl_pblas.h

Description

The p?herk routines perform a distributed matrix-matrix operation defined as

```
sub(C) := alpha*sub(A)*conjg(sub(A)') + beta*sub(C),
```

or

```
sub(C) := alpha*conjg(sub(A)')*sub(A) + beta*sub(C),
```

where:

alpha and *beta* are scalars,

sub(C) is an n -by- n Hermitian distributed matrix, *sub(C)* = $C(ic:ic+n-1, jc:jc+n-1)$.

sub(A) is a distributed matrix, *sub(A)* = $A(ia:ia+n-1, ja:ja+k-1)$, if *trans* = 'N' or 'n', and *sub(A)* = $A(ia:ia+k-1, ja:ja+n-1)$ otherwise.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix <i>sub(C)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(C)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(C)</i> is used.
<i>trans</i>	(global) CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then <i>sub(C)</i> := $\alpha\text{sub}(A)\text{conjg}(\text{sub}(A)') + \beta\text{sub}(C);$

if *trans* = 'C' or 'c', then $\text{sub}(C) := \alpha * \text{conjg}(\text{sub}(A)') * \text{sub}(A)$
+ $\beta * \text{sub}(C)$.

n (global) INTEGER. Specifies the order of the distributed matrix $\text{sub}(C)$, $n \geq 0$.

k (global) INTEGER. On entry with *trans* = 'N' or 'n', *k* specifies the number of columns of the distributed matrix $\text{sub}(A)$, and on entry with *trans* = 'T' or 't' or 'C' or 'c', *k* specifies the number of rows of the distributed matrix $\text{sub}(A)$, $k \geq 0$.

alpha (global) REAL for pcherk

DOUBLE PRECISION for pzherk

Specifies the scalar *alpha*.

a (local) COMPLEX for pcherk

DOUBLE COMPLEX for pzherk

Array, size (*lld_a*, *kla*), where *kla* is $\text{LOCq}(ja+k-1)$ when *trans* = 'N' or 'n', and is $\text{LOCq}(ja+n-1)$ otherwise. Before entry with *trans* = 'N' or 'n', this array contains the local pieces of the distributed matrix $\text{sub}(A)$.

ia, *ja* (global) INTEGER. The row and column indices in the distributed matrix *A* indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *A*.

beta (global) REAL for pcherk

DOUBLE PRECISION for pzherk

Specifies the scalar *beta*.

c (local) COMPLEX for pcherk

DOUBLE COMPLEX for pzherk

Array, size (*lld_c*, $\text{LOCq}(jc+n-1)$).

Before entry with *uplo* = 'U' or 'u', this array contains n -by- n upper triangular part of the symmetric distributed matrix $\text{sub}(C)$ and its strictly lower triangular part is not referenced.

Before entry with *uplo* = 'L' or 'l', this array contains n -by- n lower triangular part of the symmetric distributed matrix $\text{sub}(C)$ and its strictly upper triangular part is not referenced.

ic, *jc* (global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.

descC (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c

With *uplo* = 'U' or 'u', the upper triangular part of $\text{sub}(C)$ is overwritten by the upper triangular part of the updated distributed matrix.

With *uplo* = 'L' or 'l', the lower triangular part of $\text{sub}(C)$ is overwritten by the upper triangular part of the updated distributed matrix.

p?her2k

Performs a rank-2k update of a Hermitian distributed matrix.

Syntax

FORTRAN 77:

```
call pcher2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c,
ic, jc, descc)
```

```
call pzher2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c,
ic, jc, descc)
```

C:

```
void pcher2k (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , float *alpha ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , float *beta , float *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc );
```

```
void pzher2k (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , double *beta , double *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc );
```

Include Files

- C: mkl_pblas.h

Description

The p?her2k routines perform a distributed matrix-matrix operation defined as

```
sub(C):=alpha*sub(A)*conjg(sub(B)')+ conjg(alpha)*sub(B)*conjg(sub(A)')+beta*sub(C),
```

or

```
sub(C):=alpha*conjg(sub(A)')*sub(A)+ conjg(alpha)*conjg(sub(B)')*sub(A) + beta*sub(C),
```

where:

alpha and *beta* are scalars,

sub(C) is an *n*-by-*n* Hermitian distributed matrix, *sub(C)* = $C(ic:ic+n-1, jc:jc+n-1)$.

sub(A) is a distributed matrix, *sub(A)*= $A(ia:ia+n-1, ja:ja+k-1)$, if *trans* = 'N' or 'n', and *sub(A)* = $A(ia:ia+k-1, ja:ja+n-1)$ otherwise.

sub(B) is a distributed matrix, *sub(B)* = $B(ib:ib+n-1, jb:jb+k-1)$, if *trans* = 'N' or 'n', and *sub(B)*= $B(ib:ib+k-1, jb:jb+n-1)$ otherwise.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the Hermitian distributed matrix $\text{sub}(C)$ is used: If $\text{uplo} = 'U'$ or ' <u>u</u> ', then the upper triangular part of the $\text{sub}(C)$ is used. If $\text{uplo} = 'L'$ or ' <u>l</u> ', then the low triangular part of the $\text{sub}(C)$ is used.
<i>trans</i>	(global) CHARACTER*1. Specifies the operation: $\text{if } \text{trans} = 'N' \text{ or } 'n', \text{ then } \text{sub}(C) := \alpha * \text{sub}(A) * \text{conjg}(\text{sub}(B)) + \text{conjg}(\alpha) * \text{sub}(B) * \text{conjg}(\text{sub}(A)) + \beta * \text{sub}(C);$ $\text{if } \text{trans} = 'C' \text{ or } 'c', \text{ then } \text{sub}(C) := \alpha * \text{conjg}(\text{sub}(A)) * \text{sub}(A) + \text{conjg}(\alpha) * \text{conjg}(\text{sub}(B)) * \text{sub}(A) + \beta * \text{sub}(C).$
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix $\text{sub}(C)$, $n \geq 0$.
<i>k</i>	(global) INTEGER. On entry with $\text{trans} = 'N'$ or ' <u>n</u> ', <i>k</i> specifies the number of columns of the distributed matrices $\text{sub}(A)$ and $\text{sub}(B)$, and on entry with $\text{trans} = 'C'$ or ' <u>c</u> ', <i>k</i> specifies the number of rows of the distributed matrices $\text{sub}(A)$ and $\text{sub}(B)$, $k \geq 0$.
<i>alpha</i>	(global) COMPLEX for <code>pcher2k</code> DOUBLE COMPLEX for <code>pzher2k</code> Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) COMPLEX for <code>pcher2k</code> DOUBLE COMPLEX for <code>pzher2k</code> Array, size $(\text{lld}_a, \text{kla})$, where kla is $\text{LOCq}(ja+k-1)$ when $\text{trans} = 'N'$ or ' <u>n</u> ', and is $\text{LOCq}(ja+n-1)$ otherwise. Before entry with $\text{trans} = 'N'$ or ' <u>n</u> ', this array contains the local pieces of the distributed matrix $\text{sub}(A)$.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .
<i>b</i>	(local) COMPLEX for <code>pcher2k</code> DOUBLE COMPLEX for <code>pzher2k</code> Array, size $(\text{lld}_b, \text{klb})$, where klb is $\text{LOCq}(jb+k-1)$ when $\text{trans} = 'N'$ or ' <u>n</u> ', and is $\text{LOCq}(jb+n-1)$ otherwise. Before entry with $\text{trans} = 'N'$ or ' <u>n</u> ', this array contains the local pieces of the distributed matrix $\text{sub}(B)$.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.

<i>descb</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>B</i> .
<i>beta</i>	(global) REAL for pcher2k DOUBLE PRECISION for pzher2k Specifies the scalar <i>beta</i> .
<i>c</i>	(local) COMPLEX for pcher2k DOUBLE COMPLEX for pzher2k Array, size (<i>lld_c</i> , LOCq(<i>jc+n-1</i>)). Before entry with <i>uplo</i> = 'U' or 'u', this array contains <i>n</i> -by- <i>n</i> upper triangular part of the symmetric distributed matrix <i>sub(C)</i> and its strictly lower triangular part is not referenced. Before entry with <i>uplo</i> = 'L' or 'l', this array contains <i>n</i> -by- <i>n</i> lower triangular part of the symmetric distributed matrix <i>sub(C)</i> and its strictly upper triangular part is not referenced.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <i>sub(C)</i> , respectively.
<i>descC</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	With <i>uplo</i> = 'U' or 'u', the upper triangular part of <i>sub(C)</i> is overwritten by the upper triangular part of the updated distributed matrix. With <i>uplo</i> = 'L' or 'l', the lower triangular part of <i>sub(C)</i> is overwritten by the upper triangular part of the updated distributed matrix.
----------	--

p?symm

Performs a scalar-matrix-matrix product (one matrix operand is symmetric) and adds the result to a scalar-matrix product for distribute matrices.

Syntax

FORTRAN 77:

```
call pssymm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descC)
call pdsymm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descC)
call pcsymm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descC)
call pzsimm(side, uplo, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c, ic, jc, descC)
```

C:

```

void pssymm (char *side , char *uplo , MKL_INT *m , MKL_INT *n , float *alpha , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT
*jb , MKL_INT *descb , float *beta , float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT
*descc );

void pdssym (char *side , char *uplo , MKL_INT *m , MKL_INT *n , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , double *beta , double *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc );

void pcssym (char *side , char *uplo , MKL_INT *m , MKL_INT *n , float *alpha , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT
*jb , MKL_INT *descb , float *beta , float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT
*descc );

void pzssym (char *side , char *uplo , MKL_INT *m , MKL_INT *n , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , double *beta , double *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc );

```

Include Files

- C: mkl_pblas.h

Description

The p?ssymm routines perform a matrix-matrix operation with distributed matrices. The operation is defined as

```
sub(C) := alpha*sub(A)*sub(B) + beta*sub(C),
```

or

```
sub(C) := alpha*sub(B)*sub(A) + beta*sub(C),
```

where:

alpha and *beta* are scalars,

sub(A) is a symmetric distributed matrix, *sub(A)* = $A(ia:ia+m-1, ja:ja+m-1)$, if *side* = 'L', and
sub(A) = $A(ia:ia+n-1, ja:ja+n-1)$, if *side* = 'R'.

sub(B) and *sub(C)* are *m*-by-*n* distributed matrices.

sub(B) = $B(ib:ib+m-1, jb:jb+n-1)$, *sub(C)* = $C(ic:ic+m-1, jc:jc+n-1)$.

Input Parameters

<i>side</i>	(global) CHARACTER*1. Specifies whether the symmetric distributed matrix <i>sub(A)</i> appears on the left or right in the operation: if <i>side</i> = 'L' or 'l', then <i>sub(C)</i> := <i>alpha</i> * <i>sub(A)</i> * <i>sub(B)</i> + <i>beta</i> * <i>sub(C)</i> ; if <i>side</i> = 'R' or 'r', then <i>sub(C)</i> := <i>alpha</i> * <i>sub(B)</i> * <i>sub(A)</i> + <i>beta</i> * <i>sub(C)</i> .
<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix <i>sub(A)</i> is used: if <i>uplo</i> = 'U' or 'u', then the upper triangular part is used; if <i>uplo</i> = 'L' or 'l', then the lower triangular part is used.

<i>m</i>	(global) INTEGER. Specifies the number of rows of the distribute submatrix $\text{sub}(C)$, $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distribute submatrix $\text{sub}(C)$, $m \geq 0$.
<i>alpha</i>	(global) REAL for pssymm DOUBLE PRECISION for pdsymm COMPLEX for pcsymm DOUBLE COMPLEX for pzsymm Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) REAL for pssymm DOUBLE PRECISION for pdsymm COMPLEX for pcsymm DOUBLE COMPLEX for pzsymm Array, size (<i>lld_a</i> , LOCq(<i>ja+na-1</i>)). Before entry this array must contain the local pieces of the symmetric distributed matrix $\text{sub}(A)$, such that when <i>uplo</i> = 'U' or 'u', the <i>na</i> -by- <i>na</i> upper triangular part of the distributed matrix $\text{sub}(A)$ must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of $\text{sub}(A)$ is not referenced, and when <i>uplo</i> = 'L' or 'l', the <i>na</i> -by- <i>na</i> lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.
<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>b</i>	(local) REAL for pssymm DOUBLE PRECISION for pdsymm COMPLEX for pcsymm DOUBLE COMPLEX for pzsymm Array, size (<i>lld_b</i> , LOCq(<i>jb+n-1</i>)). Before entry this array must contain the local pieces of the distributed matrix $\text{sub}(B)$.
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>B</i> indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.
<i>descb</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>B</i> .
<i>beta</i>	(global) REAL for pssymm

DOUBLE PRECISION for pdssymm

COMPLEX for pcssymm

DOUBLE COMPLEX for pzssymm

Specifies the scalar *beta*.

When *beta* is set to zero, then *sub(C)* need not be set on input.

c

(local)REAL for pdssymm

DOUBLE PRECISION for pdssymm

COMPLEX for pcssymm

DOUBLE COMPLEX for pzssymm

Array, size (*lld_c*, LOC_q(*jc+n-1*)). Before entry this array must contain the local pieces of the distributed matrix *sub(C)*.

ic, jc

(global) INTEGER. The row and column indices in the distributed matrix *C* indicating the first row and the first column of the submatrix *sub(C)*, respectively.

descC

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c

Overwritten by the *m*-by-*n* updated matrix.

p?syrk

Performs a rank-*k* update of a symmetric distributed matrix.

Syntax

Fortran:

```
call pssyrk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
call pdssyrk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
call pcssyrk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
call pzssyrk(uplo, trans, n, k, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
```

C:

```
void pssyrk (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , float *alpha , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *beta , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descC );
void pdssyrk (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *beta , double *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *descC );
void pcssyrk (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , float *alpha , float
*a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *beta , float *c , MKL_INT
*ic , MKL_INT *jc , MKL_INT *descC );
```

```
void pzsyrk (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *beta , double *c ,
MKL_INT *ic , MKL_INT *jc , MKL_INT *desc );
```

Include Files

- C: mkl_pblas.h

Description

The p?syrk routines perform a distributed matrix-matrix operation defined as

```
sub(C) := alpha*sub(A)*sub(A)' + beta*sub(C),
```

or

```
sub(C) := alpha*sub(A)'*sub(A) + beta*sub(C),
```

where:

alpha and *beta* are scalars,

sub(C) is an *n*-by-*n* symmetric distributed matrix, *sub(C)* = *C*(*ic*:*ic+n-1*, *jc*:*jc+n-1*).

sub(A) is a distributed matrix, *sub(A)* = *A*(*ia*:*ia+n-1*, *ja*:*ja+k-1*), if *trans* = 'N' or 'n', and *sub(A)* = *A*(*ia*:*ia+k-1*, *ja*:*ja+n-1*) otherwise.

Input Parameters

<i>uplo</i>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix <i>sub(C)</i> is used: If <i>uplo</i> = 'U' or 'u', then the upper triangular part of the <i>sub(C)</i> is used. If <i>uplo</i> = 'L' or 'l', then the low triangular part of the <i>sub(C)</i> is used.
<i>trans</i>	(global) CHARACTER*1. Specifies the operation: if <i>trans</i> = 'N' or 'n', then <i>sub(C)</i> := <i>alpha</i> * <i>sub(A)</i> * <i>sub(A)'</i> + <i>beta</i> * <i>sub(C)</i> ; if <i>trans</i> = 'T' or 't', then <i>sub(C)</i> := <i>alpha</i> * <i>sub(A)'</i> * <i>sub(A)</i> + <i>beta</i> * <i>sub(C)</i> .
<i>n</i>	(global) INTEGER. Specifies the order of the distributed matrix <i>sub(C)</i> , <i>n</i> ≥ 0.
<i>k</i>	(global) INTEGER. On entry with <i>trans</i> = 'N' or 'n', <i>k</i> specifies the number of columns of the distributed matrix <i>sub(A)</i> , and on entry with <i>trans</i> = 'T' or 't', <i>k</i> specifies the number of rows of the distributed matrix <i>sub(A)</i> , <i>k</i> ≥ 0.
<i>alpha</i>	(global) REAL for pssyrk DOUBLE PRECISION for pdsyrk COMPLEX for pcsyrk DOUBLE COMPLEX for pzsyrk Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) REAL for pssyrk

DOUBLE PRECISION for pdssyrk

COMPLEX for pcssyrk

DOUBLE COMPLEX for pzssyrk

Array, size ($l1d_a, \ kla$), where kla is $\text{LOCq}(ja+k-1)$ when $trans = 'N'$ or ' n ', and is $\text{LOCq}(ja+n-1)$ otherwise. Before entry with $trans = 'N'$ or ' n ', this array contains the local pieces of the distributed matrix $\text{sub}(A)$.

ia, ja

(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

beta

(global)REAL for pdssyrk

DOUBLE PRECISION for pdssyrk

COMPLEX for pcssyrk

DOUBLE COMPLEX for pzssyrk

Specifies the scalar *beta*.

c

(local)REAL for pdssyrk

DOUBLE PRECISION for pdssyrk

COMPLEX for pcssyrk

DOUBLE COMPLEX for pzssyrk

Array, size ($l1d_c, \ \text{LOCq}(jc+n-1)$).

Before entry with $uplo = 'U'$ or ' u ', this array contains n -by- n upper triangular part of the symmetric distributed matrix $\text{sub}(C)$ and its strictly lower triangular part is not referenced.

Before entry with $uplo = 'L'$ or ' l ', this array contains n -by- n lower triangular part of the symmetric distributed matrix $\text{sub}(C)$ and its strictly upper triangular part is not referenced.

ic, jc

(global) INTEGER. The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.

descC

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

c

With $uplo = 'U'$ or ' u ', the upper triangular part of $\text{sub}(C)$ is overwritten by the upper triangular part of the updated distributed matrix.

With $uplo = 'L'$ or ' l ', the lower triangular part of $\text{sub}(C)$ is overwritten by the upper triangular part of the updated distributed matrix.

p?syr2k

Performs a rank- $2k$ update of a symmetric distributed matrix.

Syntax

Fortran:

```
call pssyr2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c,
ic, jc, descc)

call pdssyr2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c,
ic, jc, descc)

call pcssyr2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c,
ic, jc, descc)

call pzssyr2k(uplo, trans, n, k, alpha, a, ia, ja, desca, b, ib, jb, descb, beta, c,
ic, jc, descc)
```

C:

```
void pssyr2k (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , float *alpha ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , float *beta , float *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc );

void pdssyr2k (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , double *beta , double *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc );

void pcssyr2k (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , float *alpha ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , float *beta , float *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc );

void pzssyr2k (char *uplo , char *trans , MKL_INT *n , MKL_INT *k , double *alpha ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib ,
MKL_INT *jb , MKL_INT *descb , double *beta , double *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descc );
```

Include Files

- C: mkl_pblas.h

Description

The p?syr2k routines perform a distributed matrix-matrix operation defined as

```
sub(C) := alpha * sub(A) * sub(B)' + alpha * sub(B) * sub(A)' + beta * sub(C),
```

or

```
sub(C) := alpha * sub(A)' * sub(B) + alpha * sub(B)' * sub(A) + beta * sub(C),
```

where:

alpha and *beta* are scalars,

sub(C) is an n -by- n symmetric distributed matrix, *sub(C)=C(ic:ic+n-1, jc:jc+n-1)*.

sub(A) is a distributed matrix, *sub(A)=A(ia:ia+n-1, ja:ja+k-1)*, if *trans = 'N'* or '*n*', and *sub(A)=A(ia:ia+k-1, ja:ja+n-1)* otherwise.

`sub(B)` is a distributed matrix, `sub(B)=B(ib:ib+n-1, jb:jb+k-1)`, if `trans = 'N' or 'n'`, and `sub(B)=B(ib:ib+k-1, jb:jb+n-1)` otherwise.

Input Parameters

<code>uplo</code>	(global) CHARACTER*1. Specifies whether the upper or lower triangular part of the symmetric distributed matrix <code>sub(C)</code> is used: If <code>uplo = 'U' or 'u'</code> , then the upper triangular part of the <code>sub(C)</code> is used. If <code>uplo = 'L' or 'l'</code> , then the low triangular part of the <code>sub(C)</code> is used.
<code>trans</code>	(global) CHARACTER*1. Specifies the operation: <code>if trans = 'N' or 'n', then sub(C) := alpha*sub(A)*sub(B)' + alpha*sub(B)*sub(A)' + beta*sub(C);</code> <code>if trans = 'T' or 't', then sub(C) := alpha*sub(B)'*sub(A) + alpha*sub(A)'*sub(B) + beta*sub(C).</code>
<code>n</code>	(global) INTEGER. Specifies the order of the distributed matrix <code>sub(C)</code> , $n \geq 0$.
<code>k</code>	(global) INTEGER. On entry with <code>trans = 'N' or 'n'</code> , k specifies the number of columns of the distributed matrices <code>sub(A)</code> and <code>sub(B)</code> , and on entry with <code>trans = 'T' or 't'</code> , k specifies the number of rows of the distributed matrices <code>sub(A)</code> and <code>sub(B)</code> , $k \geq 0$.
<code>alpha</code>	(global) REAL for <code>pssyr2k</code> DOUBLE PRECISION for <code>pdsyr2k</code> COMPLEX for <code>pcsyrr2k</code> DOUBLE COMPLEX for <code>pzsyr2k</code> Specifies the scalar <code>alpha</code> .
<code>a</code>	(local) REAL for <code>pssyr2k</code> DOUBLE PRECISION for <code>pdsyr2k</code> COMPLEX for <code>pcsyrr2k</code> DOUBLE COMPLEX for <code>pzsyr2k</code> Array, size <code>(lld_a, kla)</code> , where <code>kla</code> is <code>LOCq(ja+k-1)</code> when <code>trans = 'N'</code> or <code>'n'</code> , and is <code>LOCq(ja+n-1)</code> otherwise. Before entry with <code>trans = 'N'</code> or <code>'n'</code> , this array contains the local pieces of the distributed matrix <code>sub(A)</code> .
<code>ia, ja</code>	(global) INTEGER. The row and column indices in the distributed matrix <code>A</code> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<code>desca</code>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <code>A</code> .
<code>b</code>	(local) REAL for <code>pssyr2k</code> DOUBLE PRECISION for <code>pdsyr2k</code>

COMPLEX for `pcsy2k`

DOUBLE COMPLEX for `pzsyr2k`

Array, size (lld_b, klb) , where klb is $\text{LOCq}(jb+k-1)$ when $trans = 'N'$ or ' n ', and is $\text{LOCq}(jb+n-1)$ otherwise. Before entry with $trans = 'N'$ or ' n ', this array contains the local pieces of the distributed matrix $\text{sub}(B)$.

ib, jb

(global) INTEGER. The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.

descb

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix B .

beta

(global)REAL for `pssyr2k`

DOUBLE PRECISION for `pdsyr2k`

COMPLEX for `pcsy2k`

DOUBLE COMPLEX for `pzsyr2k`

Specifies the scalar *beta*.

c

(local)REAL for `pssyr2k`

DOUBLE PRECISION for `pdsyr2k`

COMPLEX for `pcsy2k`

DOUBLE COMPLEX for `pzsyr2k`

Array, size $(lld_c, \text{LOCq}(jc+n-1))$.

Before entry with $uplo = 'U'$ or ' u ', this array contains n -by- n upper triangular part of the symmetric distributed matrix $\text{sub}(C)$ and its strictly lower triangular part is not referenced.

Before entry with $uplo = 'L'$ or ' l ', this array contains n -by- n lower triangular part of the symmetric distributed matrix $\text{sub}(C)$ and its strictly upper triangular part is not referenced.

ic, jc

(global) INTEGER. The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.

descC

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

c

With $uplo = 'U'$ or ' u ', the upper triangular part of $\text{sub}(C)$ is overwritten by the upper triangular part of the updated distributed matrix.

With $uplo = 'L'$ or ' l ', the lower triangular part of $\text{sub}(C)$ is overwritten by the upper triangular part of the updated distributed matrix.

p?tran

Transposes a real distributed matrix.

Syntax

Fortran:

```
call pstran(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
call pdtran(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
```

C:

```
void pstran (MKL_INT *m , MKL_INT *n , float *alpha , float *a , MKL_INT *ia , MKL_INT
*ja , MKL_INT *desca , float *beta , float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT
*descC );
void pdtran (MKL_INT *m , MKL_INT *n , double *alpha , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *beta , double *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descC );
```

Include Files

- C: mkl_pblas.h

Description

The p?tran routines transpose a real distributed matrix. The operation is defined as

```
sub(C) := beta*sub(C) + alpha*sub(A)'
```

where:

alpha and *beta* are scalars,

sub(C) is an *m*-by-*n* distributed matrix, *sub(C)=C(ic:ic+m-1, jc:jc+n-1)*.

sub(A) is a distributed matrix, *sub(A)=A(ia:ia+n-1, ja:ja+m-1)*.

Input Parameters

m (global) INTEGER. Specifies the number of rows of the distributed matrix
sub(C), $m \geq 0$.

n (global) INTEGER. Specifies the number of columns of the distributed matrix
sub(C), $n \geq 0$.

alpha (global) REAL for pstran
 DOUBLE PRECISION for pdtran
 Specifies the scalar *alpha*.

a (local) REAL for pstran
 DOUBLE PRECISION for pdtran
 Array, size (*l1d_a*, LOCq(*ja+m-1*)). This array contains the local pieces
 of the distributed matrix *sub(A)*.

<i>ia, ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global) REAL for pstran DOUBLE PRECISION for pdtran Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then $\text{sub}(C)$ need not be set on input.
<i>c</i>	(local) REAL for pstran DOUBLE PRECISION for pdtran Array, size (<i>l1d_c</i> , LOCq(<i>jc+n-1</i>)). This array contains the local pieces of the distributed matrix $\text{sub}(C)$.
<i>ic, jc</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.
<i>descC</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>C</i> .

Output Parameters

<i>c</i>	Overwritten by the updated submatrix.
----------	---------------------------------------

p?tranu

Transposes a distributed complex matrix.

Syntax

Fortran:

```
call pctranu(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
call pztranu(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
```

C:

```
void pctranu (MKL_INT *m , MKL_INT *n , float *alpha , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *beta , float *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descC );
void pztranu (MKL_INT *m , MKL_INT *n , double *alpha , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *beta , double *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descC );
```

Include Files

- C: mkl_pblas.h

Description

The `p?tranu` routines transpose a complex distributed matrix. The operation is defined as

```
sub(C) := beta*sub(C) + alpha*sub(A)',
```

where:

alpha and *beta* are scalars,

`sub(C)` is an *m*-by-*n* distributed matrix, `sub(C)=C(ic:ic+m-1, jc:jc+n-1)`.

`sub(A)` is a distributed matrix, `sub(A)=A(ia:ia+n-1, ja:ja+m-1)`.

Input Parameters

<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix <code>sub(C)</code> , $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix <code>sub(C)</code> , $n \geq 0$.
<i>alpha</i>	(global) COMPLEX for <code>pctransu</code> DOUBLE COMPLEX for <code>pztransu</code> Specifies the scalar <i>alpha</i> .
<i>a</i>	(local) COMPLEX for <code>pctransu</code> DOUBLE COMPLEX for <code>pztransu</code> Array, size <code>(lld_a, LOCq(ja+m-1))</code> . This array contains the local pieces of the distributed matrix <code>sub(A)</code> .
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix <code>sub(A)</code> , respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>beta</i>	(global) COMPLEX for <code>pctransu</code> DOUBLE COMPLEX for <code>pztransu</code> Specifies the scalar <i>beta</i> . When <i>beta</i> is equal to zero, then <code>sub(C)</code> need not be set on input.
<i>c</i>	(local) COMPLEX for <code>pctransu</code> DOUBLE COMPLEX for <code>pztransu</code> Array, size <code>(lld_c, LOCq(jc+n-1))</code> . This array contains the local pieces of the distributed matrix <code>sub(C)</code> .
<i>ic</i> , <i>jc</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>C</i> indicating the first row and the first column of the submatrix <code>sub(C)</code> , respectively.

descC (global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix *C*.

Output Parameters

c Overwritten by the updated submatrix.

p?tranc

Transposes a complex distributed matrix, conjugated.

Syntax

Fortran:

```
call pcstranc(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
call pzstranc(m, n, alpha, a, ia, ja, desca, beta, c, ic, jc, descC)
```

C:

```
void pcstranc (MKL_INT *m , MKL_INT *n , float *alpha , float *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , float *beta , float *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descC );
void pzstranc (MKL_INT *m , MKL_INT *n , double *alpha , double *a , MKL_INT *ia ,
MKL_INT *ja , MKL_INT *desca , double *beta , double *c , MKL_INT *ic , MKL_INT *jc ,
MKL_INT *descC );
```

Include Files

- C: mkl_pblas.h

Description

The p?tranc routines transpose a complex distributed matrix. The operation is defined as

```
sub(C):=beta*sub(C) + alpha*conjg(sub(A)'),
```

where:

alpha and *beta* are scalars,

sub(*C*) is an *m*-by-*n* distributed matrix, sub(*C*)=*C*(*ic*:*ic+m-1*, *jc*:*jc+n-1*).

sub(*A*) is a distributed matrix, sub(*A*)=*A*(*ia*:*ia+n-1*, *ja*:*ja+m-1*).

Input Parameters

m (global) INTEGER. Specifies the number of rows of the distributed matrix sub(*C*), *m* ≥ 0 .

n (global) INTEGER. Specifies the number of columns of the distributed matrix sub(*C*), *n* ≥ 0 .

alpha (global) COMPLEX for pcstranc
DOUBLE COMPLEX for pzstranc
Specifies the scalar *alpha*.

a (local) COMPLEX for pcstranc

DOUBLE COMPLEX for pztranc

Array, size (lld_a , LOCq($ja+m-1$)). This array contains the local pieces of the distributed matrix $\text{sub}(A)$.

ia, ja

(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

descA

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

beta

(global) COMPLEX for pcstranc

DOUBLE COMPLEX for pztranc

Specifies the scalar *beta*.

When *beta* is equal to zero, then $\text{sub}(C)$ need not be set on input.

c

(local) COMPLEX for pcstranc

DOUBLE COMPLEX for pztranc

Array, size (lld_c , LOCq($jc+n-1$)).

This array contains the local pieces of the distributed matrix $\text{sub}(C)$.

ic, jc

(global) INTEGER. The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.

descC

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix C .

Output Parameters

c

Overwritten by the updated submatrix.

p?trmm

Computes a scalar-matrix-matrix product (one matrix operand is triangular) for distributed matrices.

Syntax

Fortran:

```
call pstrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, descA, b, ib, jb, descB)
call pdtrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, descA, b, ib, jb, descB)
call pcstrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, descA, b, ib, jb, descB)
call pztrmm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, descA, b, ib, jb, descB)
```

C:

```
void pstrmm (char *side , char *uplo , char *transa , char *diag , MKL_INT *m ,
MKL_INT *n , float *alpha , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *descA ,
float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descB );
```

```

void pdtrmm (char *side , char *uplo , char *transa , char *diag , MKL_INT *m ,
MKL_INT *n , double *alpha , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );

void pcstrmm (char *side , char *uplo , char *transa , char *diag , MKL_INT *m ,
MKL_INT *n , float *alpha , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );

void pztrmm (char *side , char *uplo , char *transa , char *diag , MKL_INT *m ,
MKL_INT *n , double *alpha , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );

```

Include Files

- C: mkl_pblas.h

Description

The p?trmm routines perform a matrix-matrix operation using triangular matrices. The operation is defined as

```
sub(B) := alpha*op(sub(A))*sub(B)
```

or

```
sub(B) := alpha*sub(B)*op(sub(A))
```

where:

alpha is a scalar,

sub(B) is an *m*-by-*n* distributed matrix, *sub(B)*=*B*(*ib*:*ib+m-1*, *jb*:*jb+n-1*).

A is a unit, or non-unit, upper or lower triangular distributed matrix, *sub(A)*=*A*(*ia*:*ia+m-1*, *ja*:*ja+m-1*), if *side* = 'L' or 'l', and *sub(A)*=*A*(*ia*:*ia+n-1*, *ja*:*ja+n-1*), if *side* = 'R' or 'r'.

op(sub(A)) is one of *op(sub(A))* = *sub(A)*, or *op(sub(A))* = *sub(A)'*, or *op(sub(A))* = *conjg(sub(A))'*.

Input Parameters

<i>side</i>	(global) CHARACTER*1. Specifies whether <i>op(sub(A))</i> appears on the left or right of <i>sub(B)</i> in the operation: if <i>side</i> = 'L' or 'l', then <i>sub(B)</i> := <i>alpha</i> * <i>op(sub(A))</i> * <i>sub(B)</i> ; if <i>side</i> = 'R' or 'r', then <i>sub(B)</i> := <i>alpha</i> * <i>sub(B)</i> * <i>op(sub(A))</i> .
<i>uplo</i>	(global) CHARACTER*1. Specifies whether the distributed matrix <i>sub(A)</i> is upper or lower triangular: if <i>uplo</i> = 'U' or 'u', then the matrix is upper triangular; if <i>uplo</i> = 'L' or 'l', then the matrix is low triangular.
<i>transa</i>	(global) CHARACTER*1. Specifies the form of <i>op(sub(A))</i> used in the matrix multiplication: if <i>transa</i> = 'N' or 'n', then <i>op(sub(A))</i> = <i>sub(A)</i> ; if <i>transa</i> = 'T' or 't', then <i>op(sub(A))</i> = <i>sub(A)'</i> ; if <i>transa</i> = 'C' or 'c', then <i>op(sub(A))</i> = <i>conjg(sub(A))'</i> .

<i>diag</i>	(global) CHARACTER*1. Specifies whether the matrix $\text{sub}(A)$ is unit triangular: if $diag = 'U'$ or ' <u>' then the matrix is unit triangular; if $diag = 'N'$ or '<n>', then the matrix is not unit triangular.</n></u>
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix $\text{sub}(B)$, $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix $\text{sub}(B)$, $n \geq 0$.
<i>alpha</i>	(global) REAL for <code>pstrmm</code> DOUBLE PRECISION for <code>pdtrmm</code> COMPLEX for <code>pctrmm</code> DOUBLE COMPLEX for <code>pztrmm</code> Specifies the scalar <i>alpha</i> . When <i>alpha</i> is zero, then the array <i>b</i> need not be set before entry.
<i>a</i>	(local) REAL for <code>pstrmm</code> DOUBLE PRECISION for <code>pdtrmm</code> COMPLEX for <code>pctrmm</code> DOUBLE COMPLEX for <code>pztrmm</code> Array, size lld_a by <i>ka</i> , where <i>ka</i> is at least $\text{LOCq}(1, ja+m-1)$ when <i>side</i> = ' <i>L</i> ' or ' <i>l</i> ' and is at least $\text{LOCq}(1, ja+n-1)$ when <i>side</i> = ' <i>R</i> ' or ' <i>r</i> '. Before entry with <i>uplo</i> = ' <i>U</i> ' or ' <i>u</i> ', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $\text{sub}(A)$ is not referenced. Before entry with <i>uplo</i> = ' <i>L</i> ' or ' <i>l</i> ', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $\text{sub}(A)$ is not referenced . When <i>diag</i> = ' <i>U</i> ' or ' <i>u</i> ', the local entries corresponding to the diagonal elements of the submatrix $\text{sub}(A)$ are not referenced either, but are assumed to be unity.
<i>ia</i> , <i>ja</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>A</i> indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
<i>desca</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>A</i> .
<i>b</i>	(local) REAL for <code>pstrmm</code> DOUBLE PRECISION for <code>pdtrmm</code> COMPLEX for <code>pctrmm</code>

	DOUBLE COMPLEX for pztrmm Array, size (<i>lld_b</i> , LOCq(1, <i>jb+n-1</i>)).
	Before entry, this array contains the local pieces of the distributed matrix sub(<i>B</i>).
<i>ib, jb</i>	(global) INTEGER. The row and column indices in the distributed matrix <i>B</i> indicating the first row and the first column of the submatrix sub(<i>B</i>), respectively.
<i>descb</i>	(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix <i>B</i> .

Output Parameters

<i>b</i>	Overwritten by the transformed distributed matrix.
----------	--

p?trsm

Solves a distributed matrix equation (one matrix operand is triangular).

Syntax

Fortran:

```
call pstrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
call pdtrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
call pctrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
call pztrsm(side, uplo, transa, diag, m, n, alpha, a, ia, ja, desca, b, ib, jb, descb)
```

C:

```
void pstrsm (char *side , char *uplo , char *transa , char *diag , MKL_INT *m ,
MKL_INT *n , float *alpha , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );

void pdtrsm (char *side , char *uplo , char *transa , char *diag , MKL_INT *m ,
MKL_INT *n , double *alpha , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );

void pctrsm (char *side , char *uplo , char *transa , char *diag , MKL_INT *m ,
MKL_INT *n , float *alpha , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );

void pztrsm (char *side , char *uplo , char *transa , char *diag , MKL_INT *m ,
MKL_INT *n , double *alpha , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca ,
double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );
```

Include Files

- C: mkl_pblas.h

Description

The p?trsm routines solve one of the following distributed matrix equations:

$\text{op}(\text{sub}(A)) * X = \alpha * \text{sub}(B)$,

or

$X^* \text{op}(\text{sub}(A)) = \alpha * \text{sub}(B)$,

where:

α is a scalar,

X and $\text{sub}(B)$ are m -by- n distributed matrices, $\text{sub}(B) = B(ib:ib+m-1, jb:jb+n-1)$;

A is a unit, or non-unit, upper or lower triangular distributed matrix, $\text{sub}(A) = A(ia:ia+m-1, ja:ja+m-1)$, if $\text{side} = 'L'$ or ' l ', and $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$, if $\text{side} = 'R'$ or ' r ';

$\text{op}(\text{sub}(A))$ is one of $\text{op}(\text{sub}(A)) = \text{sub}(A)$, or $\text{op}(\text{sub}(A)) = \text{sub}(A)'$, or $\text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A)')$.

The distributed matrix $\text{sub}(B)$ is overwritten by the solution matrix X .

Input Parameters

<i>side</i>	(global) CHARACTER*1. Specifies whether $\text{op}(\text{sub}(A))$ appears on the left or right of X in the equation: if $\text{side} = 'L'$ or ' l ', then $\text{op}(\text{sub}(A)) * X = \alpha * \text{sub}(B)$; if $\text{side} = 'R'$ or ' r ', then $X * \text{op}(\text{sub}(A)) = \alpha * \text{sub}(B)$.
<i>uplo</i>	(global) CHARACTER*1. Specifies whether the distributed matrix $\text{sub}(A)$ is upper or lower triangular: if $\text{uplo} = 'U'$ or ' u ', then the matrix is upper triangular; if $\text{uplo} = 'L'$ or ' l ', then the matrix is low triangular.
<i>transa</i>	(global) CHARACTER*1. Specifies the form of $\text{op}(\text{sub}(A))$ used in the matrix equation: if $\text{transa} = 'N'$ or ' n ', then $\text{op}(\text{sub}(A)) = \text{sub}(A)$; if $\text{transa} = 'T'$ or ' t ', then $\text{op}(\text{sub}(A)) = \text{sub}(A)'$; if $\text{transa} = 'C'$ or ' c ', then $\text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A)')$.
<i>diag</i>	(global) CHARACTER*1. Specifies whether the matrix $\text{sub}(A)$ is unit triangular: if $\text{diag} = 'U'$ or ' u ' then the matrix is unit triangular; if $\text{diag} = 'N'$ or ' n ', then the matrix is not unit triangular.
<i>m</i>	(global) INTEGER. Specifies the number of rows of the distributed matrix $\text{sub}(B)$, $m \geq 0$.
<i>n</i>	(global) INTEGER. Specifies the number of columns of the distributed matrix $\text{sub}(B)$, $n \geq 0$.
<i>alpha</i>	(global) REAL for pstrsm DOUBLE PRECISION for pdtrsm COMPLEX for pcstrsm DOUBLE COMPLEX for pzstrsm Specifies the scalar α .

When α is zero, then a is not referenced and b need not be set before entry.

a

(local)REAL for pstrsm
 DOUBLE PRECISION for pdtrsm
 COMPLEX for pctrsm
 DOUBLE COMPLEX for pztrsm

Array, size lld_a by ka , where ka is at least $\text{LOCq}(1, ja+m-1)$ when $\text{side} = 'L'$ or ' l ' and is at least $\text{LOCq}(1, ja+n-1)$ when $\text{side} = 'R'$ or ' r '.

Before entry with $\text{uplo} = 'U'$ or ' u ', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix $\text{sub}(A)$ is not referenced.

Before entry with $\text{uplo} = 'L'$ or ' l ', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix $\text{sub}(A)$, and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix $\text{sub}(A)$ is not referenced .

When $\text{diag} = 'U'$ or ' u ', the local entries corresponding to the diagonal elements of the submatrix $\text{sub}(A)$ are not referenced either, but are assumed to be unity.

ia, ja

(global) INTEGER. The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

descA

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix A .

b

(local)REAL for pstrsm
 DOUBLE PRECISION for pdtrsm
 COMPLEX for pctrsm
 DOUBLE COMPLEX for pztrsm
 Array, size $(lld_b, \text{LOCq}(1, jb+n-1))$.

Before entry, this array contains the local pieces of the distributed matrix $\text{sub}(B)$.

ib, jb

(global) INTEGER. The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.

descB

(global and local) INTEGER array of dimension 9. The array descriptor of the distributed matrix B .

Output Parameters

b

Overwritten by the solution distributed matrix X .

Partial Differential Equations Support

12

The Intel® Math Kernel Library (Intel® MKL) provides tools for solving Partial Differential Equations (PDE). These tools are Trigonometric Transform interface routines (see [Trigonometric Transform Routines](#)) and Poisson Solver (see [Fast Poisson Solver Routines](#)).

Poisson Solver is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The solver is based on the Trigonometric Transform interface, which is, in turn, based on the Intel MKL Fast Fourier Transform (FFT) interface (refer to [Fourier Transform Functions](#)), optimized for Intel® processors.

Direct use of the Trigonometric Transform routines may be helpful to those who have already implemented their own solvers similar to the Intel MKL Poisson Solver. As it may be hard enough to modify the original code so as to make it work with Poisson Solver, you are encouraged to use fast (staggered) sine/cosine transforms implemented in the Trigonometric Transform interface to improve performance of your solver.

Both Trigonometric Transform and Poisson Solver routines can be called from C and Fortran.

NOTE

Intel MKL Trigonometric Transform and Poisson Solver routines support Fortran versions starting with Fortran 90.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Trigonometric Transform Routines

In addition to the Fast Fourier Transform (FFT) interface, described in chapter "[Fast Fourier Transforms](#)", Intel® MKL supports the Real Discrete Trigonometric Transforms (sometimes called real-to-real Discrete Fourier Transforms) interface. In this manual, the interface is referred to as TT interface. It implements a group of routines (TT routines) used to compute sine/cosine, staggered sine/cosine, and twice staggered sine/cosine transforms (referred to as staggered2 sine/cosine transforms, for brevity). The TT interface provides much flexibility of use: you can adjust routines to your particular needs at the cost of manual tuning routine parameters or just call routines with default parameter values. The current Intel MKL implementation of the TT interface can be used in solving partial differential equations and contains routines that are helpful for Fast Poisson and similar solvers.

To describe the Intel MKL TT interface, the C convention is used. Fortran users should refer to [Calling PDE Support Routines from Fortran](#).

For the list of Trigonometric Transforms currently implemented in Intel MKL TT interface, see [Transforms Implemented](#).

If you have got used to the FFTW interface (www.fftw.org), you can call the TT interface functions through real-to-real FFTW to Intel MKL wrappers without changing FFTW function calls in your code (refer to the "FFTW to Intel® MKL Wrappers for FFTW 3.x" section in [Appendix F](#) for details). However, you are strongly

encouraged to use the native TT interface for better performance. Another reason why you should use the wrappers cautiously is that TT and the real-to-real FFTW interfaces are not fully compatible and some features of the real-to-real FFTW, such as strides and multidimensional transforms, are not available through wrappers.

Transforms Implemented

TT routines allow computing the following transforms:

Forward sine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^{n-1} f(i) \sin \frac{ki\pi}{n}, k = 1, \dots, n-1$$

Backward sine transform

$$f(i) = \sum_{k=1}^{n-1} F(k) \sin \frac{ki\pi}{n}, i = 1, \dots, n-1$$

Forward staggered sine transform

$$F(k) = \frac{1}{n} \sin \frac{(2k-1)\pi}{2} f(n) + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \sin \frac{(2k-1)i\pi}{2n}, k = 1, \dots, n$$

Backward staggered sine transform

$$f(i) = \sum_{k=1}^n F(k) \sin \frac{(2k-1)i\pi}{2n}, i = 1, \dots, n$$

Forward staggered2 sine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^n f(i) \sin \frac{(2k-1)(2i-1)\pi}{4n}, k = 1, \dots, n$$

Backward staggered2 sine transform

$$f(i) = \sum_{k=1}^n F(k) \sin \frac{(2k-1)(2i-1)\pi}{4n}, i = 1, \dots, n$$

Forward cosine transform

$$F(k) = \frac{1}{n} [f(0) + f(n) \cos k\pi] + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \cos \frac{ki\pi}{n}, k = 0, \dots, n$$

Backward cosine transform

$$f(i) = \frac{1}{2} [F(0) + F(n) \cos i\pi] + \sum_{k=1}^{n-1} F(k) \cos \frac{ki\pi}{n}, i = 0, \dots, n$$

Forward staggered cosine transform

$$F(k) = \frac{1}{n} f(0) + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \cos \frac{(2k+1)i\pi}{2n}, k = 0, \dots, n-1$$

Backward staggered cosine transform

$$f(i) = \sum_{k=0}^{n-1} F(k) \cos \frac{(2k+1)i\pi}{2n}, i = 0, \dots, n-1$$

Forward staggered2 cosine transform

$$F(k) = \frac{2}{n} \sum_{i=1}^n f(i) \cos \frac{(2k-1)(2i-1)\pi}{4n}, k = 1, \dots, n$$

Backward staggered2 cosine transform

$$f(i) = \sum_{k=1}^n F(k) \cos \frac{(2k-1)(2i-1)\pi}{4n}, i = 1, \dots, n$$

NOTE

The size of the transform n can be any integer greater or equal to 2.

Sequence of Invoking TT Routines

Computation of a transform using TT interface is conceptually divided into four steps, each of which is performed via a dedicated routine. [Table "TT Interface Routines"](#) lists the routines and briefly describes their purpose and use.

Most TT routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names.

TT Interface Routines

Routine	Description
?_init_trig_transform	Initializes basic data structures of Trigonometric Transforms.

Routine	Description
?_commit_trig_transform	Checks consistency and correctness of user-defined data and creates a data structure to be used by Intel MKL FFT interface ¹ .
?_forward_trig_transform	Computes a forward/backward Trigonometric Transform of a specified type using the appropriate formula (see Transforms Implemented).
?_backward_trig_transform	Computes a forward/backward Trigonometric Transform of a specified type using the appropriate formula (see Transforms Implemented).
free_trig_transform	Releases the memory used by a data structure needed for calling FFT interface ¹ .

¹TT routines call Intel MKL FFT interface for better performance.

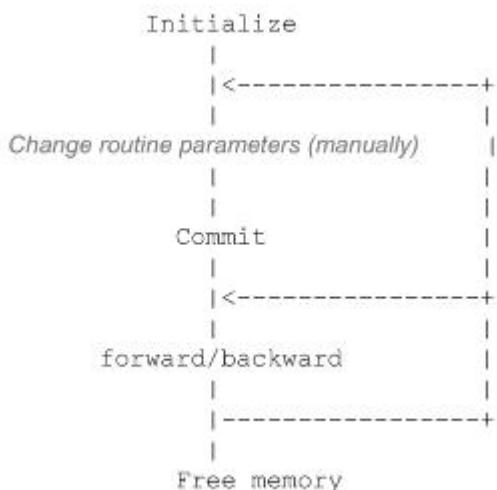
To find a transformed vector for a particular input vector only once, the Intel MKL TT interface routines are normally invoked in the order in which they are listed in [Table "TT Interface Routines"](#).

NOTE

Though the order of invoking TT routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in [Figure "Typical Order of Invoking TT Interface Routines"](#) indicates the typical order in which TT interface routines can be invoked in a general case (prefixes and suffixes in routine names are omitted).

Typical Order of Invoking TT Interface Routines



A general scheme of using TT routines for double-precision computations is shown below. A similar scheme holds for single-precision computations with the only difference in the initial letter of routine names.

```

...
    d_init_trig_transform(&n, &tt_type, ipar, dpar, &ir);
/* Change parameters in ipar if necessary. */
/* Note that the result of the Transform will be in f. If you want to preserve the data stored in f,
save it to another location before the function call below */
    d_commit_trig_transform(f, &handle, ipar, dpar, &ir);
    d_forward_trig_transform(f, &handle, ipar, dpar, &ir);
    d_backward_trig_transform(f, &handle, ipar, dpar, &ir);
    free_trig_transform(&handle, ipar, &ir);
/* here the user may clean the memory used by f, dpar, ipar */
...
  
```

You can find examples of Fortran and C code that uses TT interface routines to solve one-dimensional Helmholtz problem in the `examples\pdettf\source` and `examples\pdettc\source` folders in your Intel MKL directory.

Interface Description

All types in this documentation are either standard C types `float` and `double` or `MKL_INT` integer type. Fortran users can call the routines with `REAL` and `DOUBLE PRECISION` types of floating-point values and `INTEGER` or `INTEGER*8` integer type depending on the programming interface (LP64 or ILP64). For more information on the C types, refer to [C Datatypes Specific to Intel MKL](#) and the *Intel(R) MKL User's Guide*. To better understand usage of the types, see examples in the `examples\pdettf\source` and `examples\pdettc\source` folders in your Intel MKL directory.

Routine Options

All TT routines use parameters to pass various options to one another. These parameters are arrays `ipar`, `dpar` and `spar`. Values for these parameters should be specified very carefully (see [Common Parameters](#)). You can change these values during computations to meet your needs.

WARNING

To avoid failure or incorrect results, you must provide correct and consistent parameters to the routines.

User Data Arrays

TT routines take arrays of user data as input. For example, user arrays are passed to the routine `d_forward_trig_transform` to compute a forward Trigonometric Transform. To minimize storage requirements and improve the overall run-time efficiency, Intel MKL TT routines do not make copies of user input arrays.

NOTE

If you need a copy of your input data arrays, you must save them yourself.

For better performance, align your data arrays as recommended in the *User's Guide* (search the document for coding techniques to improve performance).

TT Routines

The section gives detailed description of TT routines, their syntax, parameters and values they return. Double-precision and single-precision versions of the same routine are described together.

TT routines call Intel MKL FFT interface (described in section "[FFT Functions](#)" in chapter "Fast Fourier Transforms"), which enhances performance of the routines.

?_init_trig_transform

Initializes basic data structures of a Trigonometric Transform.

Syntax

```
void d_init_trig_transform(MKL_INT *n, MKL_INT *tt_type, MKL_INT ipar[], double dpar[],  
MKL_INT *stat);  
  
void s_init_trig_transform(MKL_INT *n, MKL_INT *tt_type, MKL_INT ipar[], float spar[],  
MKL_INT *stat);
```

Include Files

- Fortran 90: mkl_trig_transforms.f90
- C: mkl.h

Input Parameters

<i>n</i>	MKL_INT*. Contains the size of the problem, which should be a positive integer greater than 1. Note that data vector of the transform, which other TT routines will use, must have size $n+1$ for all but staggered2 transforms. Staggered2 transforms require the vector of size n .
<i>tt_type</i>	MKL_INT*. Contains the type of transform to compute, defined via a set of named constants. The following constants are available in the current implementation of TT interface: MKL_SINE_TRANSFORM, MKL_STAGGERED_SINE_TRANSFORM, MKL_STAGGERED2_SINE_TRANSFORM; MKL_COSINE_TRANSFORM, MKL_STAGGERED_COSINE_TRANSFORM, MKL_STAGGERED2_COSINE_TRANSFORM.

Output Parameters

<i>ipar</i>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar[6]</i> . The status should be 0 to proceed to other TT routines.

Description

The `?_init_trig_transform` routine initializes basic data structures for Trigonometric Transforms of appropriate precision. After a call to `?_init_trig_transform`, all subsequently invoked TT routines use values of *ipar* and *dpar* (*spar*) array parameters returned by `?_init_trig_transform`. The routine initializes the entire array *ipar*. In the *dpar* or *spar* array, `?_init_trig_transform` initializes elements that do not depend upon the type of transform. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to the [Common Parameters](#) section. You can skip a call to the initialization routine in your code. For more information, see [Caveat on Parameter Modifications](#).

Return Values

<i>stat</i> = 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this <i>stat</i> value.
<i>stat</i> = -99999	The routine failed to complete the task.

`?_commit_trig_transform`

Checks consistency and correctness of user's data as well as initializes certain data structures required to perform the Trigonometric Transform.

Syntax

```
void d_commit_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
void s_commit_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- Fortran 90: mkl_trig_transforms.f90
- C: mkl.h

Input Parameters

<i>f</i>	double for <code>d_commit_trig_transform</code> , float for <code>s_commit_trig_transform</code> , array of size n for staggered2 transforms and of size $n+1$ for all other transforms, where n is the size of the problem. Contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors:
	<ul style="list-style-type: none"> $f[0]$ and $f[n]$ for sine transforms $f[n]$ for staggered cosine transforms $f[0]$ for staggered sine transforms. <p>Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. These restrictions meet the requirements of the Intel MKL Poisson Solver, which the TT interface is primarily designed for (for details, see Fast Poisson Solver Routines).</p>
<i>ipar</i>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.

Output Parameters

<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, refer to section " FFT Functions " in chapter "Fast Fourier Transforms").
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, $ipar[6]$ is updated with the <i>stat</i> value.
<i>dpar</i>	Contains double-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.
<i>spar</i>	Contains single-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.

`stat` MKL_INT*. Contains the routine completion status, which is also written to `ipar[6]`.

Description

The routine `?_commit_trig_transform` checks consistency and correctness of the parameters to be passed to the transform routines `?_forward_trig_transform` and/or `?_backward_trig_transform`. The routine also initializes the following data structures: `handle`, `dpar` in case of `d_commit_trig_transform`, and `spar` in case of `s_commit_trig_transform`. The `?_commit_trig_transform` routine initializes only those elements of `dpar` or `spar` that depend upon the type of transform, defined in the `?_init_trig_transform` routine and passed to `?_commit_trig_transform` with the `ipar` array. The size of the problem `n`, which determines sizes of the array parameters, is also passed to the routine with the `ipar` array and defined in the previously called `?_init_trig_transform` routine. For a detailed description of arrays `ipar`, `dpar` and `spar`, refer to the [Common Parameters](#) section. The routine performs only a basic check for correctness and consistency of the parameters. If you are going to modify parameters of TT routines, see the [Caveat on Parameter Modifications](#) section. Unlike `?_init_trig_transform`, you must call the `?_commit_trig_transform` routine in your code.

Return Values

`stat= 11`

The routine produced some warnings and made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning `ipar[6]=0` if you are sure that the parameters are correct.

`stat= 10`

The routine made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning `ipar[6]=0` if you are sure that the parameters are correct.

`stat= 1`

The routine produced some warnings. You may proceed with computations by assigning `ipar[6]=0` if you are sure that the parameters are correct.

`stat= 0`

The routine completed the task normally.

`stat= -100`

The routine stopped for any of the following reasons:

- An error in the user's data was encountered.
- Data in `ipar`, `dpar` or `spar` parameters became incorrect and/or inconsistent as a result of modifications.

`stat= -1000`

The routine stopped because of an FFT interface error.

`stat= -10000`

The routine stopped because the initialization failed to complete or the parameter `ipar[0]` was altered by mistake.

NOTE

Although positive values of *stat* usually indicate minor problems with the input data and Trigonometric Transform computations can be continued, you are highly recommended to investigate the problem first and achieve *stat*=0.

?_forward_trig_transform

Computes the forward Trigonometric Transform of type specified by the parameter.

Syntax

```
void d_forward_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
void s_forward_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- Fortran 90: mkl_trig_transforms.f90
- C: mkl.h

Input Parameters

<i>f</i>	double for <i>d_forward_trig_transform</i> , float for <i>s_forward_trig_transform</i> , array of size <i>n</i> for staggered2 transforms and of size <i>n</i> +1 for all other transforms, where <i>n</i> is the size of the problem. On input, contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors:
	<ul style="list-style-type: none"> • <i>f</i>[0] and <i>f</i>[<i>n</i>] for sine transforms • <i>f</i>[<i>n</i>] for staggered cosine transforms • <i>f</i>[0] for staggered sine transforms.
	Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. The above restrictions meet the requirements of the Intel MKL Poisson Solver, which the TT interface is primarily designed for (for details, see Fast Poisson Solver Routines).
<i>handle</i>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, see FFT Functions).
<i>ipar</i>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<i>dpar</i>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.
<i>spar</i>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.

Output Parameters

<i>f</i>	Contains the transformed vector on output.
<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar[6]</i> is updated with the <i>stat</i> value.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar[6]</i> .

Description

The routine computes the forward Trigonometric Transform of type defined in the `?_init_trig_transform` routine and passed to `?_forward_trig_transform` with the *ipar* array. The size of the problem *n*, which determines sizes of the array parameters, is also passed to the routine with the *ipar* array and defined in the previously called `?_init_trig_transform` routine. The other data that facilitates the computation is created by `?_commit_trig_transform` and supplied in *dpar* or *spar*. For a detailed description of arrays *ipar*, *dpar* and *spar*, refer to the [Common Parameters](#) section. The routine has a commit step, which calls the `?_commit_trig_transform` routine. The transform is computed according to formulas given in the [Transforms Implemented](#) section. The routine replaces the input vector *f* with the transformed vector.

NOTE

If you need a copy of the data vector *f* to be transformed, make the copy before calling the `?_forward_trig_transform` routine.

Return Values

<i>stat</i> = 0	The routine completed the task normally.
<i>stat</i> = -100	The routine stopped for any of the following reasons: <ul style="list-style-type: none"> • An error in the user's data was encountered. • Data in <i>ipar</i>, <i>dpar</i> or <i>spar</i> parameters became incorrect and/or inconsistent as a result of modifications.
<i>stat</i> = -1000	The routine stopped because of an FFT interface error.
<i>stat</i> = -10000	The routine stopped because its commit step failed to complete or the parameter <i>ipar[0]</i> was altered by mistake.

?_backward_trig_transform

Computes the backward Trigonometric Transform of type specified by the parameter.

Syntax

```
void d_backward_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);

void s_backward_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

- Fortran 90: `mkl_trig_transforms.f90`
- C: `mkl.h`

Input Parameters

<code>f</code>	<pre>double for d_backward_trig_transform,</pre> <pre>float for s_backward_trig_transform,</pre> <p>array of size n for staggered2 transforms and of size $n+1$ for all other transforms, where n is the size of the problem. On input, contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors:</p> <ul style="list-style-type: none"> $f[0]$ and $f[n]$ for sine transforms $f[n]$ for staggered cosine transforms $f[0]$ for staggered sine transforms. <p>Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. The above restrictions meet the requirements of the Intel MKL Poisson Solver, which the TT interface is primarily designed for (for details, see Fast Poisson Solver Routines).</p>
<code>handle</code>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, see FFT Functions).
<code>ipar</code>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<code>dpar</code>	double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.
<code>spar</code>	float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.

Output Parameters

<code>f</code>	Contains the transformed vector on output.
<code>ipar</code>	Contains integer data needed for Trigonometric Transform computations. On output, <code>ipar[6]</code> is updated with the <code>stat</code> value.
<code>stat</code>	MKL_INT*. Contains the routine completion status, which is also written to <code>ipar[6]</code> .

Description

The routine computes the backward Trigonometric Transform of type defined in the `?_init_trig_transform` routine and passed to `?_backward_trig_transform` with the `ipar` array. The size of the problem n , which determines sizes of the array parameters, is also passed to the routine with the `ipar` array and defined in the previously called `?_init_trig_transform` routine. The other data that facilitates the computation is created by `?_commit_trig_transform` and supplied in `dpar` or `spar`. For a detailed description of arrays `ipar`, `dpar` and `spar`, refer to the [Common Parameters](#) section. The routine

has a commit step, which calls the `?_commit_trig_transform` routine. The transform is computed according to formulas given in the [Transforms Implemented](#) section. The routine replaces the input vector f with the transformed vector.

NOTE

If you need a copy of the data vector f to be transformed, make the copy before calling the `?_backward_trig_transform` routine.

Return Values

<code>stat= 0</code>	The routine completed the task normally.
<code>stat= -100</code>	The routine stopped for any of the following reasons: <ul style="list-style-type: none"> • An error in the user's data was encountered. • Data in <code>ipar</code>, <code>dpar</code> or <code>spar</code> parameters became incorrect and/or inconsistent as a result of modifications.
<code>stat= -1000</code>	The routine stopped because of an FFT interface error.
<code>stat= -10000</code>	The routine stopped because its commit step failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.

free_trig_transform

Cleans the memory allocated for the data structure used by the FFT interface.

Syntax

```
void free_trig_transform(DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], MKL_INT *stat);
```

Include Files

- Fortran 90: `mkl_trig_transforms.f90`
- C: `mkl.h`

Input Parameters

<code>ipar</code>	MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.
<code>handle</code>	DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel MKL FFT interface (for details, refer to section " FFT Functions " in chapter "Fast Fourier Transforms").

Output Parameters

<code>handle</code>	The data structure used by Intel MKL FFT interface. Memory allocated for the structure is released on output.
---------------------	---

<i>ipar</i>	Contains integer data needed for Trigonometric Transform computations. On output, <i>ipar[6]</i> is updated with the <i>stat</i> value.
<i>stat</i>	MKL_INT*. Contains the routine completion status, which is also written to <i>ipar[6]</i> .

Description

The `free_trig_transform` routine cleans the memory used by the `handle` structure, needed for Intel MKL FFT functions. To release the memory allocated for other parameters, include cleaning of the memory in your code.

Return Values

<i>stat</i> = 0	The routine completed the task normally.
<i>stat</i> = -1000	The routine stopped because of an FFT interface error.
<i>stat</i> = -99999	The routine failed to complete the task.

Common Parameters

This section provides description of array parameters that hold TT routine options: *ipar*, *dpar* and *spar*.

NOTE

Initial values are assigned to the array parameters by the appropriate `?_init_trig_transform` and `?_commit_trig_transform` routines.

<i>ipar</i>	MKL_INT array of size 128, holds integer data needed for Trigonometric Transform computations. Its elements are described in Table "Elements of the ipar Array" :
-------------	---

Elements of the ipar Array

Index	Description
0	Contains the size of the problem to solve. The <code>?_init_trig_transform</code> routine sets <i>ipar[0]=n</i> , and all subsequently called TT routines use <i>ipar[0]</i> as the size of the transform.
1	Contains error messaging options: <ul style="list-style-type: none"> • <i>ipar[1]=-1</i> indicates that all error messages will be printed to the file <code>MKL_Trig_Transforms_log.txt</code> in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • <i>ipar[1]=0</i> indicates that no error messages will be printed. • <i>ipar[1]=1</i> (default) indicates that all error messages will be printed to the preconnected default output device (usually, screen). In case of errors, each TT routine assigns a non-zero value to <i>stat</i> regardless of the <i>ipar[1]</i> setting.
2	Contains warning messaging options:

Index	Description
	<ul style="list-style-type: none"> • <i>ipar[2]</i>=-1 indicates that all warning messages will be printed to the file <code>MKL_Trig_Transforms_log.txt</code> in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • <i>ipar[2]</i>=0 indicates that no warning messages will be printed. • <i>ipar[2]</i>=1 (default) indicates that all warning messages will be printed to the preconnected default output device (usually, screen).
	<p>In case of warnings, the <i>stat</i> parameter will acquire a non-zero value regardless of the <i>ipar[2]</i> setting.</p>
3 through 4	<p>Reserved for future use.</p>
5	<p>Contains the type of the transform. The <code>?_init_trig_transform</code> routine sets <i>ipar[5]</i>=<i>tt_type</i>, and all subsequently called TT routines use <i>ipar[5]</i> as the type of the transform.</p>
6	<p>Contains the <i>stat</i> value returned by the last completed TT routine. Used to check that the previous call to a TT routine completed with <i>stat</i>=0.</p>
7	<p>Informs the <code>?_commit_trig_transform</code> routines whether to initialize data structures <i>dpar</i> (<i>spar</i>) and <i>handle</i>. <i>ipar[7]</i>=0 indicates that the routine should skip the initialization and only check correctness and consistency of the parameters. Otherwise, the routine initializes the data structures. The default value is 1.</p>
	<p>The possibility to check correctness and consistency of input data without initializing data structures <i>dpar</i>, <i>spar</i> and <i>handle</i> enables avoiding performance losses in a repeated use of the same transform for different data vectors. Note that you can benefit from the opportunity that <i>ipar[7]</i> gives only if you are sure to have supplied proper tolerance value in the <i>dpar</i> or <i>spar</i> array. Otherwise, avoid tuning this parameter.</p>
8	<p>Contains message style options for TT routines. If <i>ipar[8]</i>=0 then TT routines print all error and warning messages in Fortran-style notations. Otherwise, TT routines print the messages in C-style notations. The default value is 1.</p>
	<p>When specifying message style options, be aware that by default, numbering of elements in C arrays starts at 0, while in Fortran arrays, it starts at 1. For example, for a C-style message "parameter <i>ipar[0]</i>=3 should be an even integer", the corresponding Fortran-style message will be "parameter <i>ipar(1)</i>=3 should be an even integer". The use of <i>ipar[8]</i> enables you to view messages in a more convenient style.</p>
9	<p>Specifies the number of OpenMP threads to run TT routines in the OpenMP environment of the Intel MKL Poisson Solver. The default value is 1. You are highly recommended not to alter this value. See also Caveat on Parameter Modifications.</p>
10	<p>Specifies the mode of compatibility with FFTW. The default value is 0. Set the value to 1 to invoke compatibility with FFTW. In the latter case, results will not be normalized, because FFTW does not do this. It is highly recommended not to alter this value, but rather use real-to-real FFTW to MKL wrappers, described in the "FFTW to Intel® MKL Wrappers for FFTW 3.x" section in Appendix F. See also Caveat on Parameter Modifications.</p>
11 through 127	<p>Reserved for future use.</p>

NOTE

While you can declare the `ipar` array as `MKL_INT ipar[11]`, for future compatibility you should declare `ipar` as `MKL_INT ipar[128]`.

Arrays `dpar` and `spar` are the same except in the data precision:

<code>dpar</code>	double array of size $5n/2+2$, holds data needed for double-precision routines to perform TT computations. This array is initialized in the <code>d_init_trig_transform</code> and <code>d_commit_trig_transform</code> routines.
<code>spar</code>	float array of size $5n/2+2$, holds data needed for single-precision routines to perform TT computations. This array is initialized in the <code>s_init_trig_transform</code> and <code>s_commit_trig_transform</code> routines.

As `dpar` and `spar` have similar elements in respective positions, the elements are described together in [Table "Elements of the dpar and spar Arrays"](#):

Elements of the dpar and spar Arrays

Index	Description
0	Contains the first absolute tolerance used by the appropriate <code>?_commit_trig_transform</code> routine. For a staggered cosine or a sine transform, <code>f[n]</code> should be equal to 0.0 and for a staggered sine or a sine transform, <code>f[0]</code> should be equal to 0.0. The <code>?_commit_trig_transform</code> routine checks whether absolute values of these parameters are below <code>dpar[0]*n</code> or <code>spar[0]*n</code> , depending on the routine precision. To suppress warnings resulting from tolerance checks, set <code>dpar[0]</code> or <code>spar[0]</code> to a sufficiently large number.
1	Reserved for future use.
2 through $5n/2+1$	Contain tabulated values of trigonometric functions. Contents of the elements depend upon the type of transform <code>tt_type</code> , set up in the <code>?_commit_trig_transform</code> routine: <ul style="list-style-type: none"> If <code>tt_type=MKL_SINE_TRANSFORM</code>, the transform uses only the first $n/2$ array elements, which contain tabulated sine values. If <code>tt_type=MKL_STAGGERED_SINE_TRANSFORM</code>, the transform uses only the first $3n/2$ array elements, which contain tabulated sine and cosine values. If <code>tt_type=MKL_STAGGERED2_SINE_TRANSFORM</code>, the transform uses all the $5n/2$ array elements, which contain tabulated sine and cosine values. If <code>tt_type=MKL_COSINE_TRANSFORM</code>, the transform uses only the first n array elements, which contain tabulated cosine values. If <code>tt_type=MKL_STAGGERED_COSINE_TRANSFORM</code>, the transform uses only the first $3n/2$ elements, which contain tabulated sine and cosine values. If <code>tt_type=MKL_STAGGERED2_COSINE_TRANSFORM</code>, the transform uses all the $5n/2$ elements, which contain tabulated sine and cosine values.

NOTE

To save memory, you can define the array size depending upon the type of transform.

Caveat on Parameter Modifications

Flexibility of the TT interface enables you to skip a call to the `?_init_trig_transform` routine and to initialize the basic data structures explicitly in your code. You may also need to modify the contents of `ipar`, `dpar` and `spar` arrays after initialization. When doing so, provide correct and consistent data in the arrays.

Mistakenly altered arrays cause errors or wrong computation. You can perform a basic check for correctness and consistency of parameters by calling the `?_commit_trig_transform` routine; however, this does not ensure the correct result of a transform but only reduces the chance of errors or wrong results.

NOTE

To supply correct and consistent parameters to TT routines, you should have considerable experience in using the TT interface and good understanding of elements that the `ipar`, `spar` and `dpar` arrays contain and dependencies between values of these elements.

However, in rare occurrences, even advanced users might fail to compute a transform using TT routines after the parameter modifications. In cases like these, refer for technical support at <http://www.intel.com/software/products/support/>.

WARNING

The only way that ensures proper computation of the Trigonometric Transforms is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of `ipar`, `dpar` and `spar` arrays unless a strong need arises.

Implementation Details

Several aspects of the Intel MKL TT interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, Intel MKL provides you with the TT language-specific header files to include in your code:

- `mkl_trig_transforms.h`, to be used together with `mkl_dfti.h`, for C programs.
- `mkl_trig_transforms.f90`, to be used together with `mkl_dfti.f90`, for Fortran programs.

NOTE

- Intel MKL TT interface supports Fortran versions starting with Fortran 90.
 - Use of the Intel MKL TT software without including the above language-specific header files is not supported.
-

C-specific Header File

The C-specific header file below defines the following function prototypes:

```
void d_init_trig_transform(MKL_INT *, MKL_INT *, MKL_INT *, double *, MKL_INT *);
void d_commit_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, double *, MKL_INT *);
void d_forward_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, double *, MKL_INT *);
void d_backward_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, double *, MKL_INT *);

void s_init_trig_transform(MKL_INT *, MKL_INT *, MKL_INT *, float *, MKL_INT *);
void s_commit_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, float *, MKL_INT *);
void s_forward_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, float *, MKL_INT *);
void s_backward_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, float *, MKL_INT *);

void free_trig_transform(DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, MKL_INT *);
```

Fortran-Specific Header File

The Fortran-Specific header file below defines the following function prototypes:

```
SUBROUTINE D_INIT_TRIG_TRANSFORM(n, tt_type, ipar, dpar, stat)
  INTEGER, INTENT(IN) :: n, tt_type
  INTEGER, INTENT(INOUT) :: ipar(*)
```

```

REAL(8), INTENT(INOUT) :: dpar(*)
INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_INIT_TRIG_TRANSFORM

SUBROUTINE D_COMMIT_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
  REAL(8), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(8), INTENT(INOUT) :: dpar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_COMMIT_TRIG_TRANSFORM

SUBROUTINE D_FORWARD_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
  REAL(8), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(8), INTENT(INOUT) :: dpar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_FORWARD_TRIG_TRANSFORM

SUBROUTINE D_BACKWARD_TRIG_TRANSFORM(f, handle, ipar, dpar, stat)
  REAL(8), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(8), INTENT(INOUT) :: dpar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE D_BACKWARD_TRIG_TRANSFORM

SUBROUTINE S_INIT_TRIG_TRANSFORM(n, tt_type, ipar, spar, stat)
  INTEGER, INTENT(IN) :: n, tt_type
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(4), INTENT(INOUT) :: spar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_INIT_TRIG_TRANSFORM

SUBROUTINE S_COMMIT_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
  REAL(4), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(4), INTENT(INOUT) :: spar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_COMMIT_TRIG_TRANSFORM

SUBROUTINE S_FORWARD_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
  REAL(4), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(4), INTENT(INOUT) :: spar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_FORWARD_TRIG_TRANSFORM

SUBROUTINE S_BACKWARD_TRIG_TRANSFORM(f, handle, ipar, spar, stat)
  REAL(4), INTENT(INOUT) :: f(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: handle
  INTEGER, INTENT(INOUT) :: ipar(*)
  REAL(4), INTENT(INOUT) :: spar(*)
  INTEGER, INTENT(OUT) :: stat
END SUBROUTINE S_BACKWARD_TRIG_TRANSFORM

SUBROUTINE FREE_TRIG_TRANSFORM(handle, ipar, stat)

```

```

INTEGER, INTENT(INOUT) :: ipar(*)
TYPE(DFTI_DESCRIPTOR), POINTER :: handle
INTEGER, INTENT(OUT) :: stat
END SUBROUTINE FREE_TRIG_TRANSFORM

```

Fortran specifics of the TT routines usage are similar for all Intel MKL PDE support tools and described in the [Calling PDE Support Routines from Fortran](#) section.

Fast Poisson Solver Routines

In addition to the Real Discrete Trigonometric Transforms (TT) interface (refer to [Trigonometric Transform Routines](#)), Intel MKL supports the the Poisson Solver interface. This interface implements a group of routines (Poisson Solver routines) used to compute a solution of Laplace, Poisson, and Helmholtz problems of a special kind using discrete Fourier transforms. Laplace and Poisson problems are special cases of a more general Helmholtz problem. The problems that are solved by the Poisson Solver interface are defined more exactly in the [Poisson Solver Implementation](#) subsection. The Poisson Solver interface provides much flexibility of use: you can call routines with the default parameter values or adjust routines to your particular needs by manually tuning routine parameters. You can adjust the style of error and warning messages to a C or Fortran notation by setting up a dedicated parameter. This adds convenience to debugging, because you can read information in the way that is natural for your code. The Intel MKL Poisson Solver interface currently contains only routines that implement the following solvers:

- Fast Laplace, Poisson and Helmholtz solvers in a Cartesian coordinate system
- Fast Poisson and Helmholtz solvers in a spherical coordinate system.

To describe the Intel MKL Poisson Solver interface, the C convention is used. Fortran usage specifics can be found in the [Calling PDE Support Routines from Fortran](#) section.

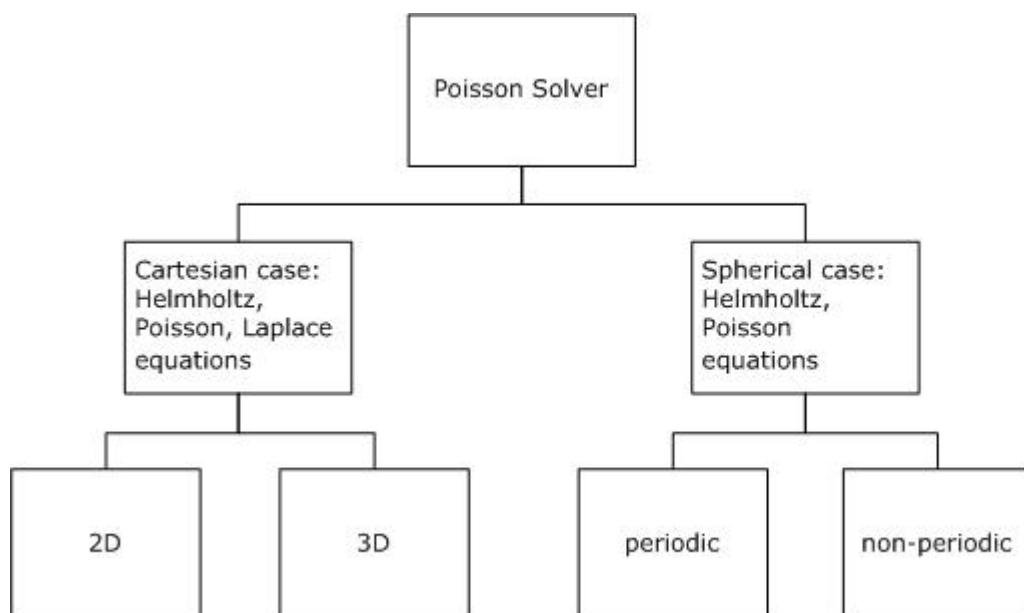
NOTE

Fortran users should keep in mind that array indices in Fortran start at 1 instead of 0, as they do in C.

Poisson Solver Implementation

Poisson Solver routines enable approximate solving of certain two-dimensional and three-dimensional problems. [Figure "Structure of the Poisson Solver"](#) shows the general structure of the Poisson Solver.

Structure of the Poisson Solver



NOTE

Although in the Cartesian case, both periodic and non-periodic solvers are also supported, they use the same interfaces.

Sections below provide details of the problems that can be solved using Intel MKL Poisson Solver.

Two-Dimensional Problems

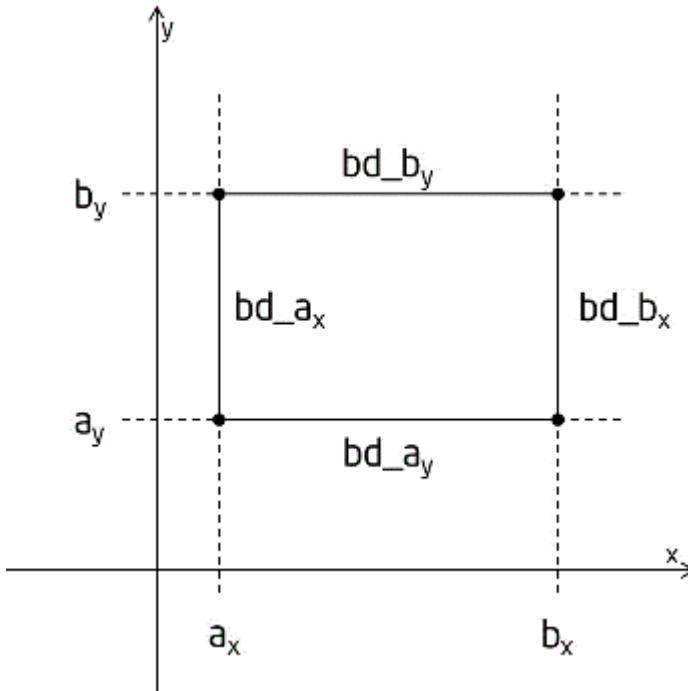
Notational Conventions

The Poisson Solver interface description uses the following notation for boundaries of a rectangular domain $a_x < x < b_x, a_y < y < b_y$ on a Cartesian plane:

$$bd_a_x = \{x = a_x, a_y \leq y \leq b_y\}, bd_b_x = \{x = b_x, a_y \leq y \leq b_y\}$$

$$bd_a_y = \{a_x \leq x \leq b_x, y = a_y\}, bd_b_y = \{a_x \leq x \leq b_x, y = b_y\}.$$

The following figure shows these boundaries:



The wildcard "+" may stand for any of the symbols a_x, b_x, a_y, b_y , so bd_+ denotes any of the above boundaries.

The Poisson Solver interface description uses the following notation for boundaries of a rectangular domain $a_\phi < \phi < b_\phi, a_\theta < \theta < b_\theta$ on a sphere $0 \leq \phi \leq 2\pi, 0 \leq \theta \leq \pi$:

$$bd_a_\phi = \{\phi = a_\phi, a_\theta \leq \theta \leq b_\theta\}, bd_b_\phi = \{\phi = b_\phi, a_\theta \leq \theta \leq b_\theta\}$$

$$bd_a_\theta = \{a_\phi \leq \phi \leq b_\phi, \theta = a_\theta\}, bd_b_\theta = \{a_\phi \leq \phi \leq b_\phi, \theta = b_\theta\}.$$

The wildcard "~" may stand for any of the symbols $a_\phi, b_\phi, a_\theta, b_\theta$, so $bd_~$ denotes any of the above boundaries.

Two-dimensional Helmholtz problem on a Cartesian plane

The two-dimensional (2D) Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + qu = f(x, y), q = \text{const} \geq 0$$

in a rectangle, that is, a rectangular domain $a_x < x < b_x, a_y < y < b_y$, with one of the following boundary conditions on each boundary bd_+ :

- The Dirichlet boundary condition

$$u(x, y) = G(x, y)$$

- The Neumann boundary condition

$$\frac{\partial u}{\partial n}(x, y) = g(x, y)$$

where

$n = -x$ on bd_a_x , $n = x$ on bd_b_x ,

$n = -y$ on bd_a_y , $n = y$ on bd_b_y .

- Periodic boundary conditions

$$u(a_x, y) = u(b_x, y), \frac{\partial}{\partial x} u(a_x, y) = \frac{\partial}{\partial x} u(b_x, y),$$

$$u(x, a_y) = u(x, b_y), \frac{\partial}{\partial y} u(x, a_y) = \frac{\partial}{\partial y} u(x, b_y)$$

Two-dimensional Poisson problem on a Cartesian plane

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The 2D Poisson problem is to find an approximate solution of the Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

in a rectangle $a_x < x < b_x, a_y < y < b_y$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_+ . In case of a problem with the Neumann boundary condition on the entire boundary, you can find the solution of the problem only up to a constant. In this case, the Poisson Solver will compute the solution that provides the minimal Euclidean norm of a residual.

Two-dimensional (2D) Laplace problem on a Cartesian plane

The Laplace problem is a special case of the Helmholtz problem, when $q=0$ and $f(x, y)=0$. The 2D Laplace problem is to find an approximate solution of the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

in a rectangle $a_x < x < b_x, a_y < y < b_y$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_{-+} .

Helmholtz problem on a sphere

The Helmholtz problem on a sphere is to find an approximate solution of the Helmholtz equation

$$-\Delta_s u + qu = f, \quad q = const \geq 0,$$

$$\Delta_s = \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \phi^2} + \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left(\sin \theta \frac{\partial}{\partial \theta} \right)$$

in a domain bounded by angles $a_\phi \leq \phi \leq b_\phi, a_\theta \leq \theta \leq b_\theta$ (spherical rectangle), with boundary conditions for particular domains listed in [Table "Details of Helmholtz Problem on a Sphere"](#).

Details of Helmholtz Problem on a Sphere

Domain on a sphere	Boundary condition	Periodic/non-periodic case
Rectangular, that is, $b_\phi - a_\phi < 2\pi$ and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on each boundary bd_{-+}	non-periodic
Where $a_\phi = 0, b_\phi = 2\pi$, and $b_\theta - a_\theta < \pi$	Homogeneous Dirichlet boundary conditions on the boundaries bd_{-a_θ} and bd_{-b_θ}	periodic
Entire sphere, that is, $a_\phi = 0, b_\phi = 2\pi, a_\theta = 0$, and $b_\theta = \pi$	Boundary condition $\left(\sin \theta \frac{\partial u}{\partial \theta} \right)_{\theta \rightarrow 0}^{0 \rightarrow \pi} = 0$	periodic

at the poles.

Poisson problem on a sphere

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The Poisson problem on a sphere is to find an approximate solution of the Poisson equation

$$-\Delta_s u = f, \quad \Delta_s = \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \phi^2} + \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left(\sin \theta \frac{\partial}{\partial \theta} \right)$$

in a spherical rectangle $a_\phi \leq \phi \leq b_\phi, a_\theta \leq \theta \leq b_\theta$ in cases listed in [Table "Details of Helmholtz Problem on a Sphere"](#). The solution to the Poisson problem on the entire sphere can be found up to a constant only. In this case, Poisson Solver will compute the solution that provides the minimal Euclidean norm of a residual.

Approximation of 2D problems

To find an approximate solution for any of the 2D problems, in the rectangular domain a uniform mesh can be defined for the Cartesian case as:

$$\begin{aligned} & \left\{ x_i = a_x + i h_x, y_j = a_y + j h_y \right\}, \\ & i = 0, \dots, n_x, j = 0, \dots, n_y, h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y} \end{aligned}$$

and for the spherical case as:

$$\begin{aligned} & \left\{ \varphi_i = a_\varphi + i h_\varphi, \theta_j = a_\theta + j h_\theta \right\}, \\ & i = 0, \dots, n_\varphi, j = 0, \dots, n_\theta, h_\varphi = \frac{b_\varphi - a_\varphi}{n_\varphi}, h_\theta = \frac{b_\theta - a_\theta}{n_\theta} \end{aligned}$$

The Poisson Solver uses the standard five-point finite difference approximation on this mesh to compute the approximation to the solution:

- In the Cartesian case, the values of the approximate solution will be computed in the mesh points (x_i, y_j) provided that you can supply the values of the right-hand side $f(x, y)$ in these points and the values of the appropriate boundary functions $G(x, y)$ and/or $g(x, y)$ in the mesh points laying on the boundary of the rectangular domain.
- In the spherical case, the values of the approximate solution will be computed in the mesh points (φ_i, θ_j) provided that you can supply the values of the right-hand side $f(\varphi, \theta)$ in these points.

NOTE

The number of mesh intervals n_φ in the φ direction of a spherical mesh must be even in the periodic case. The Poisson Solver does not support spherical meshes that do not meet this condition.

Three-Dimensional Problems

Notational Conventions

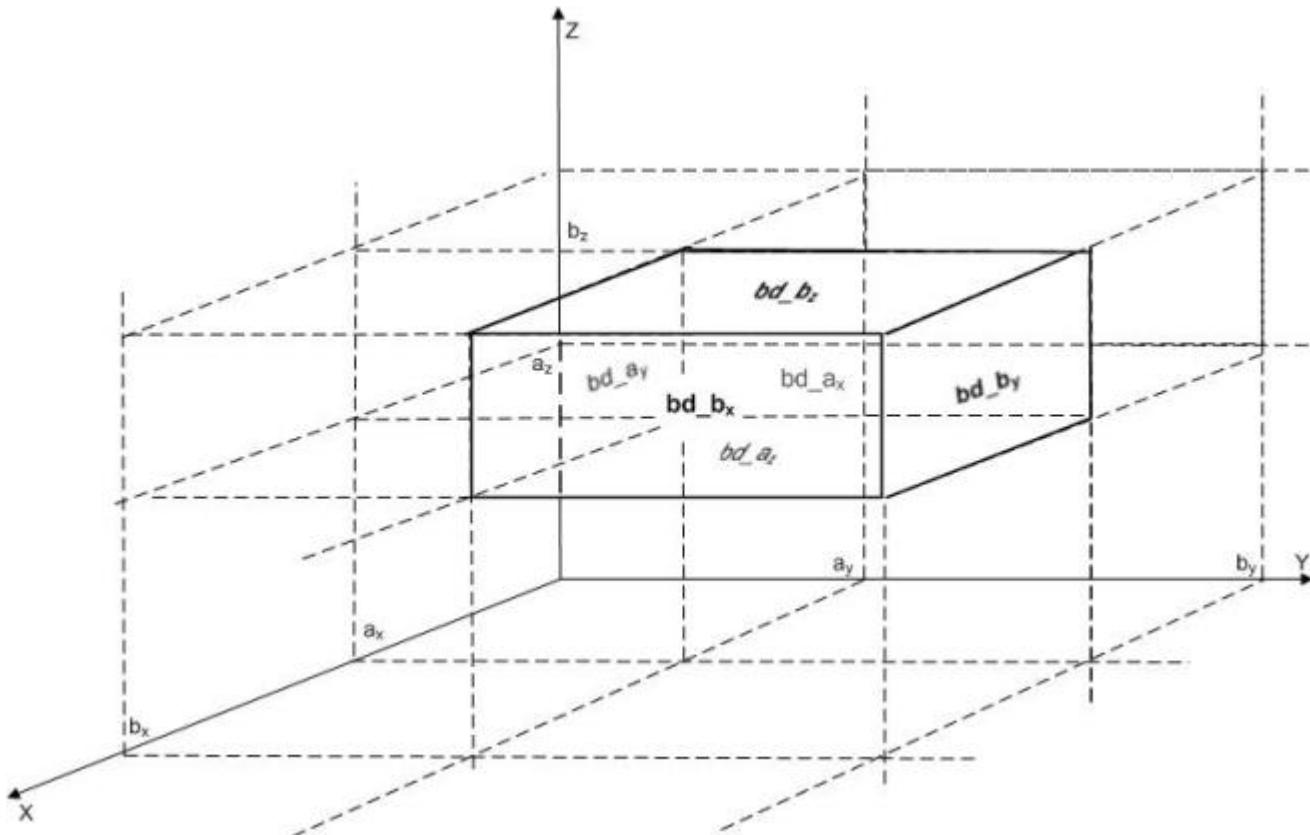
The Poisson Solver interface description uses the following notation for boundaries of a parallelepiped domain $a_x < x < b_x, a_y < y < b_y, a_z < z < b_z$:

$$bd_a_x = \{x = a_x, a_y \leq y \leq b_y, a_z \leq z \leq b_z\}, \quad bd_b_x = \{x = b_x, a_y \leq y \leq b_y, a_z \leq z \leq b_z\}$$

$$bd_a_y = \{a_x \leq x \leq b_x, y = a_y, a_z \leq z \leq b_z\}, \quad bd_b_y = \{a_x \leq x \leq b_x, y = b_y, a_z \leq z \leq b_z\}$$

$$bd_a_z = \{a_x \leq x \leq b_x, a_y \leq y \leq b_y, z = a_z\}, \quad bd_b_z = \{a_x \leq x \leq b_x, a_y \leq y \leq b_y, z = b_z\}.$$

The following figure shows these boundaries:



The wildcard "+" may stand for any of the symbols $a_x, b_x, a_y, b_y, a_z, b_z$, so bd_+ denotes any of the above boundaries.

Three-dimensional (3D) Helmholtz problem

The 3D Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} + qu = f(x, y, z), q = \text{const} \geq 0$$

in a parallelepiped, that is, a parallelepiped domain $a_x < x < b_x, a_y < y < b_y, a_z < z < b_z$, with one of the following boundary conditions on each boundary bd_+ :

- The Dirichlet boundary condition

$$u(x, y, z) = G(x, y, z)$$

- The Neumann boundary condition

$$\frac{\partial u}{\partial n}(x, y, z) = g(x, y, z)$$

where

$$n = -x \text{ on } bd_a_x, n = x \text{ on } bd_b_x,$$

$n = -y$ on bd_a_y , $n = y$ on bd_b_y ,

$n = -z$ on bd_a_z , $n = z$ on bd_b_z .

- Periodic boundary conditions

$$u(a_x, y, z) = u(b_x, y, z), \frac{\partial}{\partial x} u(a_x, y, z) = \frac{\partial}{\partial x} u(b_x, y, z),$$

$$u(x, a_y, z) = u(x, b_y, z), \frac{\partial}{\partial y} u(x, a_y, z) = \frac{\partial}{\partial y} u(x, b_y, z),$$

$$u(x, y, a_z) = u(x, y, b_z), \frac{\partial}{\partial z} u(x, y, a_z) = \frac{\partial}{\partial z} u(x, y, b_z)$$

Three-dimensional (3D) Poisson problem

The Poisson problem is a special case of the Helmholtz problem, when $q=0$. The 3D Poisson problem is to find an approximate solution of the Poisson equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f(x, y, z)$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_+ .

Three-dimensional (3D) Laplace problem

The Laplace problem is a special case of the Helmholtz problem, when $q=0$ and $f(x, y, z)=0$. The 3D Laplace problem is to find an approximate solution of the Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0$$

in a parallelepiped $a_x < x < b_x$, $a_y < y < b_y$, $a_z < z < b_z$ with the Dirichlet, Neumann, or periodic boundary conditions on each boundary bd_+ .

Approximation of 3D problems

To find an approximate solution for each of the 3D problems, a uniform mesh can be defined in the parallelepiped domain as:

$$\{x_i = a_x + i h_x, y_j = a_y + j h_y, z_k = a_z + k h_z\}$$

where

$$i = 0, \dots, n_x, j = 0, \dots, n_y, k = 0, \dots, n_z$$

$$h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}, h_z = \frac{b_z - a_z}{n_z}$$

The Poisson Solver uses the standard seven-point finite difference approximation on this mesh to compute the approximation to the solution. The values of the approximate solution will be computed in the mesh points (x_i, y_j, z_k) , provided that you can supply the values of the right-hand side $f(x, y, z)$ in these points and the values of the appropriate boundary functions $G(x, y, z)$ and/or $g(x, y, z)$ in the mesh points laying on the boundary of the parallelepiped domain.

Sequence of Invoking Poisson Solver Routines

NOTE

This description always shows the solution process for the Helmholtz problem, because Fast Poisson Solvers and Fast Laplace Solvers are special cases of Fast Helmholtz Solvers (see [Poisson Solver Implementation](#)).

The Poisson Solver interface enables you to compute a solution of the Helmholtz problem in four steps. Each step is performed by a dedicated routine. [Table "Poisson Solver Interface Routines"](#) lists the routines and briefly describes their purpose.

Most Poisson Solver routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "`s`" and "`d`". The wildcard "?" stands for either of these symbols in routine names. The routines for the Cartesian coordinate system have 2D and 3D versions. Their names end respectively in "`2D`" and "`3D`". The routines for spherical coordinate system have periodic and non-periodic versions. Their names end respectively in "`p`" and "`np`".

Poisson Solver Interface Routines

Routine	Description
<code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/</code> <code>?_init_sph_p/?_init_sph_np</code>	Initializes basic data structures for Fast Helmholtz Solver in the 2D/3D/periodic/non-periodic case, respectively.
<code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/</code> <code>?_commit_sph_p/?_commit_sph_np</code>	Checks consistency and correctness of input data and initializes data structures for the solver, including those used by the Intel MKL FFT interface ¹ .
<code>?_Helmholtz_2D/?_Helmholtz_3D/ ?_sph_p/?_sph_np</code>	Computes an approximate solution of the 2D/3D/periodic/non-periodic Helmholtz problem (see Poisson Solver Implementation) specified by the parameters.
<code>free_Helmholtz_2D/free_Helmholtz_3D/</code> <code>free_sph_p/free_sph_np</code>	Releases the memory used by the data structures needed for calling the Intel MKL FFT interface ¹ .

¹ Poisson Solver routines call the Intel MKL FFT interface for better performance.

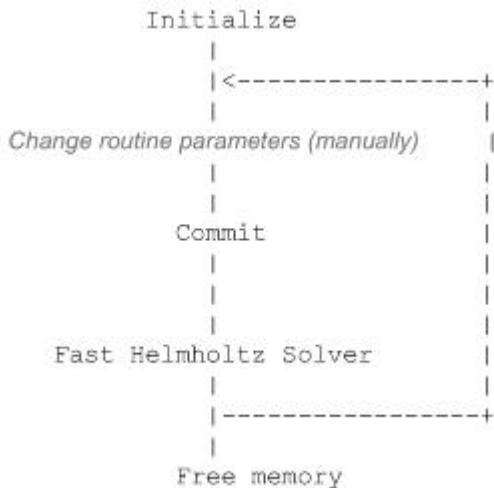
To find an approximate solution of Helmholtz problem only once, the Intel MKL Poisson Solver interface routines are normally invoked in the order in which they are listed in [Table "Poisson Solver Interface Routines"](#).

NOTE

Though the order of invoking Poisson Solver routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in [Figure "Typical Order of Invoking Poisson Solver Routines"](#) indicates the typical order in which Poisson Solver routines can be invoked in a general case.

[Typical Order of Invoking Poisson Solver Routines](#)



A general scheme of using Poisson Solver routines for double-precision computations in a 3D Cartesian case is shown below. You can change this scheme to a scheme for single-precision computations by changing the initial letter of the Poisson Solver routine names from "d" to "s". You can also change the scheme below from the 3D to 2D case by changing the ending of the Poisson Solver routine names.

```

...
d_init_Helmholtz_3D(&ax, &bx, &ay, &by, &az, &bz, &nx, &ny, &nz, BCtype, ipar, dpar, &stat);
/* change parameters in ipar and/or dpar if necessary. */
/* note that the result of the Fast Helmholtz Solver will be in f. If you want to keep the data
that is stored in f, save it to another location before the function call below */
d_commit_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar, dpar,
&stat);
d_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar, dpar, &stat);
free_Helmholtz_3D (&xhandle, &yhandle, ipar, &stat);
/* here you may clean the memory used by f, dpar, ipar */
...

```

A general scheme of using Poisson Solver routines for double-precision computations in a spherical periodic case is shown below. You can change this scheme to a scheme for single-precision computations by changing the initial letter of the Poisson Solver routine names from "d" to "s". You can also change the scheme below to a scheme for a non-periodic case by changing the ending of the Poisson Solver routine names from "p" to "np".

```

...
d_init_sph_p(&ap,&bp,&at,&bt,&np,&nt,&q,ipar,dpar,&stat);
/* change parameters in ipar and/or dpar if necessary. */
/* note that the result of the Fast Helmholtz Solver will be in f. If you want to keep the data
that is stored in f, save it to another location before the function call below */
d_commit_sph_p(f,&handle_s,&handle_c,ipar,dpar,&stat);
d_sph_p(f,&handle_s,&handle_c,ipar,dpar,&stat);
free_sph_p(&handle_s,&handle_c,ipar,&stat);
/* here you may clean the memory used by f, dpar, ipar */
...

```

You can find examples of Fortran and C code that uses Poisson Solver routines to solve Helmholtz problem (in both Cartesian and spherical cases) in the `examples\pdepoissonf\source` and `examples\pdepoissons\source` folders in your Intel MKL directory.

Interface Description

All numerical types in this section are either standard C types `float` and `double` or `MKL_INT` integer type. Fortran users can call the routines with `REAL` and `DOUBLE PRECISION` types of floating-point values and either `INTEGER` or `INTEGER*8` integer type depending on the programming interface (LP64 or ILP64). For more information on the C types, refer to [C Datatypes Specific to Intel MKL](#) and the [Intel\(R\) MKL User's Guide](#). To better understand usage of the types, see examples in the `examples\pdepoissonf\source` and `examples\pdepoissons\source` folders in your Intel MKL directory.

Routine Options

All Poisson Solver routines use parameters for passing various options to the routines. These parameters are arrays `ipar`, `dpar`, and `spar`. Values for these parameters should be specified very carefully (see [Common Parameters](#)). You can change these values during computations to meet your needs. For more details, see the descriptions of specific routines.

WARNING

To avoid failure or incorrect results, you must provide correct and consistent parameters to the routines.

User Data Arrays

Poisson Solver routines take arrays of user data as input. For example, the `d_Helmholtz_3D` routine takes user arrays to compute an approximate solution to the 3D Helmholtz problem. To minimize storage requirements and improve the overall run-time efficiency, Intel MKL Poisson Solver routines do not make copies of user input arrays.

NOTE

If you need a copy of your input data arrays, you must save them yourself.

For better performance, align your data arrays as recommended in the *User's Guide* (search the document for coding techniques to improve performance).

Routines for the Cartesian Solver

The section describes Poisson Solver routines for the Cartesian case, their syntax, parameters, and return values. All flavors of the same routine are described together: single- and double-precision and 2D and 3D.

NOTE

Some of the routine parameters are used only in the 3D Fast Helmholtz Solver.

Poisson Solver routines call Intel MKL FFT routines (described in section "[FFT Functions](#)" in chapter "Fast Fourier Transforms"), which enhance performance of the Poisson Solver routines.

[?_init_Helmholtz_2D/?_init_Helmholtz_3D](#)

Initializes basic data structures of the Fast 2D/3D Helmholtz Solver.

Syntax

```

void d_init_Helmholtz_2D(double* ax, double* bx, double* ay, double* by, MKL_INT* nx,
                           MKL_INT* ny, char* BCtype, double* q, MKL_INT* ipar, double* dpar, MKL_INT* stat);
void s_init_Helmholtz_2D(float* ax, float* bx, float* ay, float* by, MKL_INT* nx,
                           MKL_INT* ny, char* BCtype, float* q, MKL_INT* ipar, float* spar, MKL_INT* stat);
void d_init_Helmholtz_3D(double* ax, double* bx, double* ay, double* by, double* az,
                           double* bz, MKL_INT* nx, MKL_INT* ny, MKL_INT* nz, char* BCtype, double* q, MKL_INT* ipar,
                           double* dpar, MKL_INT* stat);
void s_init_Helmholtz_3D(float* ax, float* bx, float* ay, float* by, float* az, float* bz,
                           MKL_INT* nx, MKL_INT* ny, MKL_INT* nz, char* BCtype, float* q, MKL_INT* ipar,
                           float* spar, MKL_INT* stat);

```

Include Files

- Fortran 90: mkl_poisson.f90
- C: mkl.h

Input Parameters

ax double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D,
 float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
 The coordinate of the leftmost boundary of the domain along the x-axis.

bx double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D,
 float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
 The coordinate of the rightmost boundary of the domain along the x-axis.

ay double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D,
 float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
 The coordinate of the leftmost boundary of the domain along the y-axis.

by double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D,
 float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D.
 The coordinate of the rightmost boundary of the domain along the y-axis.

az double* for d_init_Helmholtz_3D,
 float* for s_init_Helmholtz_3D.
 The coordinate of the leftmost boundary of the domain along the z-axis.
 This parameter is needed only for the ?_init_Helmholtz_3D routine.

bz double* for d_init_Helmholtz_3D,
 float* for s_init_Helmholtz_3D.
 The coordinate of the rightmost boundary of the domain along the z-axis.
 This parameter is needed only for the ?_init_Helmholtz_3D routine.

nx MKL_INT*. The number of mesh intervals along the x-axis.

ny MKL_INT*. The number of mesh intervals along the y-axis.

<i>nz</i>	MKL_INT*. The number of mesh intervals along the z-axis. This parameter is needed only for the ?_init_Helmholtz_3D routine.
<i>BCtype</i>	char*. Contains the type of boundary conditions on each boundary. Must contain four characters for ?_init_Helmholtz_2D and six characters for ?_init_Helmholtz_3D. Each of the characters can be 'N' (Neumann boundary condition), 'D' (Dirichlet boundary condition), or 'P' (periodic boundary conditions). Specify the types of boundary conditions for the boundaries in the following order: <i>bd_ax</i> , <i>bd_bx</i> , <i>bd_ay</i> , <i>bd_by</i> , <i>bd_az</i> , and <i>bd_bz</i> . Specify periodic boundary conditions on the respective boundaries in pairs (for example, 'PPDD' or 'NNPP' in the 2D case). The types of boundary conditions for the last two boundaries are needed only in the 3D case.
<i>q</i>	<pre>double* for d_init_Helmholtz_2D/d_init_Helmholtz_3D, float* for s_init_Helmholtz_2D/s_init_Helmholtz_3D .</pre> <p>The constant Helmholtz coefficient. Note that to solve Poisson or Laplace problem, you should set the value of <i>q</i> to 0.</p>

Output Parameters

<i>ipar</i>	MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to ipar).
<i>dpar</i>	double array of size $5*nx/2+7$ in the 2D case or $5*(nx+ny)/2+9$ in the 3D case. Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).
<i>spar</i>	float array of size $5*nx/2+7$ in the 2D case or $5*(nx+ny)/2+9$ in the 3D case. Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> . Continue to call other Poisson Solver routines only if the status is 0.

Description

The ?_init_Helmholtz_2D/?_init_Helmholtz_3D routines initialize basic data structures for Poisson Solver computations of the appropriate precision. All routines invoked after a call to a ?_init_Helmholtz_2D/?_init_Helmholtz_3D routine use values of the *ipar*, *dpar* and *spar* array parameters returned by the routine. Detailed description of the array parameters can be found in [Common Parameters](#).

CAUTION

Data structures initialized and created by 2D flavors of the routine cannot be used by 3D flavors of any Poisson Solver routines, and vice versa.

You can skip calls to these routines in your code. However, see [Caveat on Parameter Modifications](#) for information on initializing the data structures.

Return Values

<code>stat= 0</code>	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this <code>stat</code> value.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

[_commit_Helmholtz_2D/?_commit_Helmholtz_3D](#)

Checks consistency and correctness of input data and initializes certain data structures required to solve 2D/3D Helmholtz problem.

Syntax

```
void d_commit_Helmholtz_2D(double* f, double* bd_ax, double* bd_bx, double* bd_ay,
double* bd_by, DFTI_DESCRIPTOR_HANDLE* xhandle, MKL_INT* ipar, double* dpar, MKL_INT* stat);

void s_commit_Helmholtz_2D(float* f, float* bd_ax, float* bd_bx, float* bd_ay, float*
bd_by, DFTI_DESCRIPTOR_HANDLE* xhandle, MKL_INT* ipar, float* spar, MKL_INT* stat);

void d_commit_Helmholtz_3D(double* f, double* bd_ax, double* bd_bx, double* bd_ay,
double* bd_by, double* bd_az, double* bd_bz, DFTI_DESCRIPTOR_HANDLE* xhandle,
DFTI_DESCRIPTOR_HANDLE* yhandle, MKL_INT* ipar, double* dpar, MKL_INT* stat);

void s_commit_Helmholtz_3D(float* f, float* bd_ax, float* bd_bx, float* bd_ay, float*
bd_by, float* bd_az, float* bd_bz, DFTI_DESCRIPTOR_HANDLE* xhandle,
DFTI_DESCRIPTOR_HANDLE* yhandle, MKL_INT* ipar, float* spar, MKL_INT* stat);
```

Include Files

- Fortran 90: `mkl_poisson.f90`
- C: `mkl.h`

Input Parameters

<code>f</code>	<code>double*</code> for <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code> , <code>float*</code> for <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code> . Contains the right-hand side of the problem packed in a single vector:
	<ul style="list-style-type: none"> • 2D problem: The size of the vector for the is $(nx+1)*(ny+1)$. The value of the right-hand side in the mesh point (i, j) is stored in $f[i+j*(nx+1)]$. • 3D problem: The size of the vector for the is $(nx+1)*(ny+1)*(nz+1)$. The value of the right-hand side in the mesh point (i, j, k) is stored in $f[i+j*(nx+1)+k*(nx+1)*(ny+1)]$.
	Note that to solve the Laplace problem, you should set all the elements of the array <code>f</code> to 0.
	Note also that the array <code>f</code> may be altered by the routine. To preserve the <code>f</code> vector, save it to another memory location.

<code>ipar</code>	<code>MKL_INT</code> array of size 128. Contains integer data to be used by the Fast Helmholtz Solver (for details, refer to ipar).
-------------------	--

<i>dpar</i>	double array of size depending on the dimension of the problem:
	<ul style="list-style-type: none"> • 2D problem: $5*nx/2+7$ • 3D problem: $5*(nx+ny)/2+9$
	Contains double-precision data to be used by the Fast Helmholtz Solver (for details, refer to dpar and spar).
<i>spar</i>	float array of size depending on the dimension of the problem:
	<ul style="list-style-type: none"> • 2D problem: $5*nx/2+7$ • 3D problem: $5*(nx+ny)/2+9$
	Contains single-precision data to be used by the Fast Helmholtz Solver (for details, refer to dpar and spar).
<i>bd_ax</i>	double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D, float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.
	Contains values of the boundary condition on the leftmost boundary of the domain along the <i>x</i> -axis (for more information, refer to a detailed description of bd_ax).
<i>bd_bx</i>	double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D, float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.
	Contains values of the boundary condition on the rightmost boundary of the domain along the <i>x</i> -axis (for more information, refer to a detailed description of bd_bx).
<i>bd_ay</i>	double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D, float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.
	Contains values of the boundary condition on the leftmost boundary of the domain along the <i>y</i> -axis (for more information, refer to a detailed description of bd_ay).
<i>bd_by</i>	double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D, float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.
	Contains values of the boundary condition on the rightmost boundary of the domain along the <i>y</i> -axis (for more information, refer to a detailed description of bd_by).
<i>bd_az</i>	double* for d_commit_Helmholtz_3D, float* for s_commit_Helmholtz_3D.
	Used only by ?_commit_Helmholtz_3D. Contains values of the boundary condition on the leftmost boundary of the domain along the <i>z</i> -axis (for more information, refer to a detailed description of bd_az).
<i>bd_bz</i>	double* for d_commit_Helmholtz_3D, float* for s_commit_Helmholtz_3D.

Used only by `?_commit_Helmholtz_3D`. Contains values of the boundary condition on the rightmost boundary of the domain along the z-axis (for more information, refer to [a detailed description of `bd_bz`](#)).

Output Parameters

<code>f</code>	Contains right-hand side of the problem, possibly altered on output.
<code>ipar</code>	Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in ipar .
<code>dpar</code>	Contains double-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in dpar and spar .
<code>spar</code>	Contains single-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in dpar and spar .
<code>xhandle, yhandle</code>	<code>DFTI_DESCRIPTOR_HANDLE*</code> . Data structures used by the Intel MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). <code>yhandle</code> is used only by <code>?_commit_Helmholtz_3D</code> .
<code>stat</code>	<code>MKL_INT*</code> . Routine completion status, which is also written to <code>ipar[0]</code> . Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routines check the consistency and correctness of the parameters to be passed to the solver routines `?_Helmholtz_2D/?_Helmholtz_3D`. They also initialize the `xhandle` and `yhandle` data structures, `ipar` array, and `dpar` or `spar` array, depending upon the routine precision. Refer to [Common Parameters](#) to find out which particular array elements the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routines initialize and to what values these elements are initialized.

The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of Poisson Solver routines, see [Caveat on Parameter Modifications](#).

Unlike `?_init_Helmholtz_2D/?_init_Helmholtz_3D`, you must call

the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routines in your code. Values of `ax, bx, ay, by, az,` and `bz` are passed to the routines with the `spar/dpar` array, and values of `nx, ny, nz`, and `BCType` are passed with the `ipar` array.

Return Values

<code>stat= 1</code>	The routine completed without errors but with warnings.
<code>stat= 0</code>	The routine successfully completed the task.
<code>stat= -100</code>	The routine stopped because an error in the input data was found, or the data in the <code>dpar, spar</code> , or <code>ipar</code> array was altered by mistake.
<code>stat= -1000</code>	The routine stopped because of an Intel MKL FFT or TT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.

`stat= -99999`

The routine failed to complete the task because of a fatal error.

?_Helmholtz_2D/?_Helmholtz_3D

Computes the solution of the 2D/3D Helmholtz problem specified by the parameters.

Syntax

```
void d_Helmholtz_2D(double* f, double* bd_ax, double* bd_bx, double* bd_ay, double* bd_by, DFTI_DESCRIPTOR_HANDLE* xhandle, MKL_INT* ipar, double* dpar, MKL_INT* stat);
void s_Helmholtz_2D(float* f, float* bd_ax, float* bd_bx, float* bd_ay, float* bd_by, DFTI_DESCRIPTOR_HANDLE* xhandle, MKL_INT* ipar, float* spar, MKL_INT* stat);
void d_Helmholtz_3D(double* f, double* bd_ax, double* bd_bx, double* bd_ay, double* bd_by, double* bd_az, double* bd_bz, DFTI_DESCRIPTOR_HANDLE* xhandle, DFTI_DESCRIPTOR_HANDLE* yhandle, MKL_INT* ipar, double* dpar, MKL_INT* stat);
void s_Helmholtz_3D(float* f, float* bd_ax, float* bd_bx, float* bd_ay, float* bd_by, float* bd_az, float* bd_bz, DFTI_DESCRIPTOR_HANDLE* xhandle, DFTI_DESCRIPTOR_HANDLE* yhandle, MKL_INT* ipar, float* spar, MKL_INT* stat);
```

Include Files

- Fortran 90: `mkl_poisson.f90`
- C: `mkl.h`

Input Parameters

`f`

`double* for d_Helmholtz_2D/d_Helmholtz_3D,`
`float* for s_Helmholtz_2D/s_Helmholtz_3D.`

Contains the right-hand side of the problem packed in a single vector and modified by the appropriate `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine. Note that an attempt to substitute the original right-hand side vector, which was passed to the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine, at this point results in an incorrect solution.

- 2D problem: the size of the vector is $(nx+1)*(ny+1)$. The value of the modified right-hand side in the mesh point (i, j) is stored in $f[i+j*(nx+1)]$.
- 3D problem: the size of the vector is $(nx+1)*(ny+1)*(nz+1)$. The value of the modified right-hand side in the mesh point (i, j, k) is stored in $f[i+j*(nx+1)+k*(nx+1)*(ny+1)]$.

`xhandle, yhandle`

`DFTI_DESCRIPTOR_HANDLE*`. Data structures used by the Intel MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). `yhandle` is used only by `?_Helmholtz_3D`.

`ipar`

`MKL_INT` array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to `ipar`).

`dpar`

`double` array of size depending on the dimension of the problem:

- 2D problem: $5*nx/2+7$

- 3D problem: $5*(nx+ny)/2+9$

Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to [dpar](#) and [spar](#)).

spar

float array of size depending on the dimension of the problem:

- 2D problem: $5*nx/2+7$
- 3D problem: $5*(nx+ny)/2+9$

Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to [dpar](#) and [spar](#)).

bd_ax

double* for d_Helmholtz_2D/d_Helmholtz_3D,
float* for s_Helmholtz_2D/s_Helmholtz_3D.

Contains values of the boundary condition on the leftmost boundary of the domain along the x-axis (for more information, refer to [a detailed description of bd_ax](#)).

bd_bx

double* for d_Helmholtz_2D/d_Helmholtz_3D,
float* for s_Helmholtz_2D/s_Helmholtz_3D.

Contains values of the boundary condition on the rightmost boundary of the domain along the x-axis (for more information, refer to [a detailed description of bd_bx](#)).

bd_ay

double* for d_Helmholtz_2D/d_Helmholtz_3D,
float* for s_Helmholtz_2D/s_Helmholtz_3D.

Contains values of the boundary condition on the leftmost boundary of the domain along the y-axis for more information, refer to [a detailed description of bd_ay](#)).

bd_by

double* for d_Helmholtz_2D/d_Helmholtz_3D,
float* for s_Helmholtz_2D/s_Helmholtz_3D.

Contains values of the boundary condition on the rightmost boundary of the domain along the y-axis (for more information, refer to [a detailed description of bd_by](#)).

bd_az

double* for d_Helmholtz_3D,
float* for s_Helmholtz_3D.

Used only by ?_Helmholtz_3D. Contains values of the boundary condition on the leftmost boundary of the domain along the z-axis (for more information, refer to [a detailed description of bd_az](#)).

bd_bz

double* for d_Helmholtz_3D,
float* for s_Helmholtz_3D.

Used only by ?_Helmholtz_3D. Contains values of the boundary condition on the rightmost boundary of the domain along the z-axis (for more information, refer to [a detailed description of bd_bz](#)).

NOTE

To avoid incorrect computation results, do not change arrays `bd_ax`, `bd_bx`, `bd_ay`, `bd_by`, `bd_az`, `bd_bz` between a call to the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine and a subsequent call to the appropriate `?_Helmholtz_2D/?_Helmholtz_3D` routine.

Output Parameters

<code>f</code>	On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.
<code>xhandle</code> , <code>yhandle</code>	Data structures used by the Intel MKL FFT interface. Although the addresses do not change, the structures are modified on output.
<code>ipar</code>	Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in <code>ipar</code> .
<code>stat</code>	<code>MKL_INT*</code> . Routine completion status, which is also written to <code>ipar[0]</code> . Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_Helmholtz_2D/?_Helmholtz_3D` routines compute the approximate solution of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to formulas given in the [Poisson Solver Implementation](#) section. The `f` parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of `ax`, `bx`, `ay`, `by`, `az`, and `bz` are passed to the routines with the `spar/dpar` array, and values of `nx`, `ny`, `nz`, and `BCtype` are passed with the `ipar` array.

Return Values

<code>stat= 1</code>	The routine completed without errors but with some warnings.
<code>stat= 0</code>	The routine successfully completed the task.
<code>stat= -2</code>	The routine stopped because division by zero occurred. It usually happens if the data in the <code>dpar</code> or <code>spar</code> array was altered by mistake.
<code>stat= -3</code>	The routine stopped because the sufficient memory was unavailable for the computations.
<code>stat= -100</code>	The routine stopped because an error in the input data was found or the data in the <code>dpar</code> , <code>spar</code> , or <code>ipar</code> array was altered by mistake.
<code>stat= -1000</code>	The routine stopped because of the Intel MKL FFT or TT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

free_Helmholtz_2D/free_Helmholtz_3D

Releases the memory allocated for the data structures used by the FFT interface.

Syntax

```
void free_Helmholtz_2D(DFTI_DESCRIPTOR_HANDLE* xhandle, MKL_INT* ipar, MKL_INT* stat);
void free_Helmholtz_3D(DFTI_DESCRIPTOR_HANDLE* xhandle, DFTI_DESCRIPTOR_HANDLE* yhandle, MKL_INT* ipar, MKL_INT* stat);
```

Include Files

- Fortran 90: mkl_poisson.f90
- C: mkl.h

Input Parameters

<i>xhandle</i> , <i>yhandle</i>	DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). The structure <i>yhandle</i> is used only by free_Helmholtz_3D.
<i>ipar</i>	MKL_INT array of size 128. Contains integer data used by Fast Helmholtz Solver (for details, refer to <i>ipar</i>).

Output Parameters

<i>xhandle</i> , <i>yhandle</i>	Data structures used by the Intel MKL FFT interface. Memory allocated for the structures is released on output.
<i>ipar</i>	Contains integer data used by Fast Helmholtz Solver. On output, the status of the routine call is written to <i>ipar[0]</i> .
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> .

Description

The free_Helmholtz_2D-free_Helmholtz_3D routine releases the memory used by the *xhandle* and *yhandle* structures, which are needed for calling the Intel MKL FFT functions. To release memory allocated for other parameters, include memory release statements in your code.

Return Values

<i>stat</i> = 0	The routine successfully completed the task.
<i>stat</i> = -1000	The routine stopped because of an Intel MKL FFT or TT interface error.
<i>stat</i> = -99999	The routine failed to complete the task because of a fatal error.

Routines for the Spherical Solver

The section describes Poisson Solver routines for the spherical case, their syntax, parameters, and return values. All flavors of the same routine are described together: single- and double-precision and periodic (having names ending in "p") and non-periodic (having names ending in "np").

These Poisson Solver routines also call the Intel MKL FFT routines (described in section "FFT Functions" in chapter "Fast Fourier Transforms"), which enhance the performance of the Poisson Solver routines.

?_init_sph_p/?_init_sph_np

Initializes basic data structures of the periodic and non-periodic Fast Helmholtz Solver on a sphere.

Syntax

```
void d_init_sph_p(double* ap, double* at, double* bp, double* bt, MKL_INT* np, MKL_INT* nt, double* q, MKL_INT* ipar, double* dpar, MKL_INT* stat);
void s_init_sph_p(float* ap, float* at, float* bp, float* bt, MKL_INT* np, MKL_INT* nt, float* q, MKL_INT* ipar, float* spar, MKL_INT* stat);
void d_init_sph_np(double* ap, double* at, double* bp, double* bt, MKL_INT* np, MKL_INT* nt, double* q, MKL_INT* ipar, double* dpar, MKL_INT* stat);
void s_init_sph_np(float* ap, float* at, float* bp, float* bt, MKL_INT* np, MKL_INT* nt, float* q, MKL_INT* ipar, float* spar, MKL_INT* stat);
```

Include Files

- Fortran 90: mkl_poisson.f90
- C: mkl.h

Input Parameters

<i>ap</i>	double* for d_init_sph_p/d_init_sph_np, float* for s_init_sph_p/s_init_sph_np.
	The coordinate (angle) of the leftmost boundary of the domain along the ϕ -axis.
<i>bp</i>	double* for d_init_sph_p/d_init_sph_np, float* for s_init_sph_p/s_init_sph_np.
	The coordinate (angle) of the rightmost boundary of the domain along the ϕ -axis.
<i>at</i>	double* for d_init_sph_p/d_init_sph_np, float* for s_init_sph_p/s_init_sph_np.
	The coordinate (angle) of the leftmost boundary of the domain along the θ -axis.
<i>bt</i>	double* for d_init_sph_p/d_init_sph_np, float* for s_init_sph_p/s_init_sph_np.
	The coordinate (angle) of the rightmost boundary of the domain along the θ -axis.
<i>np</i>	MKL_INT*. The number of mesh intervals along the ϕ -axis. Must be even in the periodic case.
<i>nt</i>	MKL_INT*. The number of mesh intervals along the θ -axis.
<i>q</i>	double* for d_init_sph_p/d_init_sph_np,

```
float* for s_init_sph_p/s_init_sph_np.
```

The constant Helmholtz coefficient. To solve the Poisson problem, set the value of q to 0.

Output Parameters

<i>ipar</i>	MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to ipar).
<i>dpar</i>	double array of size $5*np/2+nt+10$. Contains double-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
<i>spar</i>	float array of size $5*np/2+nt+10$. Contains single-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> . Continue to call other Poisson Solver routines only if the status is 0.

Description

The `?_init_sph_p/?_init_sph_np` routines initialize basic data structures for Poisson Solver computations. All routines invoked after a call to a `?_init_Helmholtz_2D/?_init_Helmholtz_3D` routine use values of the *ipar*, *dpar*, and *spar* array parameters returned by the routine. A detailed description of the array parameters can be found in [Common Parameters](#).

CAUTION

Data structures initialized and created by periodic flavors of the routine cannot be used by non-periodic flavors of any Poisson Solver routines for Helmholtz Solver on a sphere, and vice versa.

You can skip calls to these routines in your code. However, see [Caveat on Parameter Modifications](#) for information on initializing the data structures.

Return Values

<i>stat</i> = 0	The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this <i>stat</i> value.
<i>stat</i> = -99999	The routine failed to complete the task because of fatal error.

[?_commit_sph_p/?_commit_sph_np](#)

Checks consistency and correctness of input data and initializes certain data structures required to solve the periodic/non-periodic Helmholtz problem on a sphere.

Syntax

```
void d_commit_sph_p(double* f, DFTI_DESCRIPTOR_HANDLE* handle_s,
DFTI_DESCRIPTOR_HANDLE* handle_c, MKL_INT* ipar, double* dpar, MKL_INT* stat);
```

```
void s_commit_sph_p(float* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE** handle_c, MKL_INT* ipar, float* spar, MKL_INT* stat);
void d_commit_sph_np(double* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, double* dpar, MKL_INT* stat);
void s_commit_sph_np(float* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, float* spar, MKL_INT* stat);
```

Include Files

- Fortran 90: mkl_poisson.f90
- C: mkl.h

Input Parameters

<i>f</i>	double* for <code>d_commit_sph_p/d_commit_sph_np</code> , float* for <code>s_commit_sph_p/s_commit_sph_np</code> .
	Contains the right-hand side of the problem packed in a single vector. The size of the vector is $(np+1)*(nt+1)$ and value of the right-hand side in the mesh point (i, j) is stored in $f[i+j*(np+1)]$.
	Note that the array <i>f</i> may be altered by the routine. Save this vector to another memory location if you want to preserve it.
<i>ipar</i>	MKL_INT array of size 128. Contains integer data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to ipar).
<i>dpar</i>	double array of size $5*np/2+nt+10$. Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).
<i>spar</i>	float array of size $5*np/2+nt+10$. Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).

Output Parameters

<i>f</i>	Contains the right-hand side of the problem, possibly altered on output.
<i>ipar</i>	Contains integer data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in ipar .
<i>dpar</i>	Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar .
<i>spar</i>	Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in dpar and spar .
<i>handle_s</i> , <i>handle_c</i> , <i>handle</i> DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). <i>handle_s</i> and <i>handle_c</i> are used only in <code>?_commit_sph_p</code> and <i>handle</i> is used only in <code>?_commit_sph_np</code> .	
<i>stat</i> MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> . Continue to call other Poisson Solver routines only if the status is 0.	

Description

The `?_commit_sph_p/?_commit_sph_np` routines check consistency and correctness of the parameters to be passed to the solver routines `?_sph_p/?_sph_np`, respectively. They also initialize certain data structures. The routine `?_commit_sph_p` initializes structures `handle_s` and `handle_c`, and `?_commit_sph_np` initializes `handle`. The routines also initialize the `ipar` array and `dpar` or `spar` array, depending upon the routine precision. Refer to [Common Parameters](#) to find out which particular array elements the `?_commit_sph_p/?_commit_sph_np` routines initialize and to what values these elements are initialized.

The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of Poisson Solver routines, see [Caveat on Parameter Modifications](#).

Unlike `?_init_sph_p/?_init_sph_np`, you must call the `?_commit_sph_p/?_commit_sph_np` routines. Values of `np` and `nt` are passed to each of the routines with the `ipar` array.

Return Values

<code>stat= 1</code>	The routine completed without errors but with warnings.
<code>stat= 0</code>	The routine successfully completed the task.
<code>stat= -100</code>	The routine stopped because an error in the input data was found or the data in the <code>dpar</code> , <code>spar</code> , or <code>ipar</code> array was altered by mistake.
<code>stat= -1000</code>	The routine stopped because of an Intel MKL FFT or TT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

`?_sph_p/?_sph_np`

Computes the solution of the spherical Helmholtz problem specified by the parameters.

Syntax

```
void d_sph_p(double* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE* handle_c, MKL_INT* ipar, double* dpar, MKL_INT* stat);

void s_sph_p(float* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE* handle_c, MKL_INT* ipar, float* spar, MKL_INT* stat);

void d_sph_np(double* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, double* dpar, MKL_INT* stat);

void s_sph_np(float* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, float* spar, MKL_INT* stat);
```

Include Files

- Fortran 90: `mkl_poisson.f90`
- C: `mkl.h`

Input Parameters

<i>f</i>	<pre>double* for d_sph_p/d_sph_np, float* for s_sph_p/s_sph_np.</pre> <p>Contains the right-hand side of the problem packed in a single vector and modified by the appropriate <code>?_commit_sph_p/?_commit_sph_np</code> routine. Note that an attempt to substitute the original right-hand side vector, which was passed to the <code>?_commit_sph_p/?_commit_sph_np</code> routine, at this point results in an incorrect solution.</p> <p>The size of the vector is $(np+1)*(nt+1)$ and the value of the modified right-hand side in the mesh point (i, j) is stored in <code>f[i+j*(np+1)]</code>.</p>
<i>handle_s</i> , <i>handle_c</i> , <i>handleDFTI_DESCRIPTOR_HANDLE*</i>	Data structures used by Intel MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). <i>handle_s</i> and <i>handle_c</i> are used only in <code>?_sph_p</code> and <i>handle</i> is used only in <code>?_sph_np</code> .
<i>ipar</i>	<code>MKL_INT</code> array of size 128. Contains integer data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to <code>ipar</code>).
<i>dpar</i>	<code>double</code> array of size $5*np/2+nt+10$. Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to <code>dpar</code> and <code>spar</code>).
<i>spar</i>	<code>float</code> array of size $5*np/2+nt+10$. Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to <code>dpar</code> and <code>spar</code>).

Output Parameters

<i>f</i>	On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.
<i>handle_s</i> , <i>handle_c</i> , <i>handleData</i>	Data structures used by the Intel MKL FFT interface.
<i>ipar</i>	Contains integer data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in <code>ipar</code> .
<i>dpar</i>	Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in <code>dpar</code> and <code>spar</code> .
<i>spar</i>	Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in <code>dpar</code> and <code>spar</code> .
<i>stat</i>	<code>MKL_INT*</code> . Routine completion status, which is also written to <code>ipar[0]</code> . Continue to call other Poisson Solver routines only if the status is 0.

Description

The `sph_p/sph_np` routines compute the approximate solution on a sphere of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to the formulas given in the [Poisson Solver Implementation](#) section. The `f` parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of `np` and `nt` are passed to each of the routines with the `ipar` array.

Return Values

<code>stat= 1</code>	The routine completed without errors but with warnings.
<code>stat= 0</code>	The routine successfully completed the task.
<code>stat= -2</code>	The routine stopped because division by zero occurred. It usually happens if the data in the <code>dpar</code> or <code>spar</code> array was altered by mistake.
<code>stat= -3</code>	The routine stopped because the memory was insufficient to complete the computations.
<code>stat= -100</code>	The routine stopped because an error in the input data was found or the data in the <code>dpar</code> , <code>spar</code> , or <code>ipar</code> array was altered by mistake.
<code>stat= -1000</code>	The routine stopped because of an Intel MKL FFT or TT interface error.
<code>stat= -10000</code>	The routine stopped because the initialization failed to complete or the parameter <code>ipar[0]</code> was altered by mistake.
<code>stat= -99999</code>	The routine failed to complete the task because of a fatal error.

free_sph_p/free_sph_np

Releases the memory allocated for the data structures used by the FFT interface.

Syntax

```
void free_sph_p(DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE* handle_c,
                 MKL_INT* ipar, MKL_INT* stat);

void free_sph_np(DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, MKL_INT* stat);
```

Include Files

- Fortran 90: `mkl_poisson.f90`
- C: `mkl.h`

Input Parameters

`handle_s`, `handle_c`, `handle` `DFTI_DESCRIPTOR_HANDLE*`. Data structures used by the Intel MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). The structures `handle_s` and `handle_c` are used only in `free_sph_p`, and `handle` is used only in `free_sph_np`.

`ipar` `MKL_INT` array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to `ipar`).

Output Parameters

<i>handle_s</i> , <i>handle_c</i> , <i>handle</i>	Data structures used by the Intel MKL FFT interface. Memory allocated for the structures is released on output.
<i>ipar</i>	Contains integer data to be used by Fast Helmholtz Solver on a sphere. On output, the status of the routine call is written to <i>ipar[0]</i> .
<i>stat</i>	MKL_INT*. Routine completion status, which is also written to <i>ipar[0]</i> .

Description

The `free_sph_p/free_sph_np` routine releases the memory used by the `handle_s`, `handle_c` or `handle` structures, needed for calling the Intel MKL FFT functions. To release memory allocated for other parameters, include memory release statements in your code.

Return Values

<i>stat</i> = 0	The routine successfully completed the task.
<i>stat</i> = -1000	The routine stopped because of an Intel MKL FFT or TT interface error.
<i>stat</i> = -99999	The routine failed to complete the task because of a fatal error.

Common Parameters

ipar

<i>ipar</i>	MKL_INT array of size 128, holds integer data needed for Fast Helmholtz Solver (both for Cartesian and spherical coordinate systems). Its elements are described in Table "Elements of the ipar Array" :
-------------	--

NOTE

Initial values are assigned to the array parameters by the appropriate `?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np` and `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np` routines.

Elements of the ipar Array

Index	Description
0	Contains status value of the last Poisson Solver routine called. In general, it should be 0 on exit from a routine to proceed with the Fast Helmholtz Solver. The element has no predefined values. This element can also be used to inform the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np</code> routines of how the Commit step of the computation should be carried out (see Figure "Typical Order of Invoking Poisson Solver Routines"). A non-zero value of <i>ipar[0]</i> with decimal representation

$$\overline{abc} = 100a + 10b + c$$

Index	Description
<p>=100a+10b+c, where each of a, b, and c is equal to 0 or 9, indicates that some parts of the Commit step should be omitted.</p>	
<ul style="list-style-type: none"> • If c=9, the routine omits checking of parameters and initialization of the data structures. • If b=9, <ul style="list-style-type: none"> • In the Cartesian case, the routine omits the adjustment of the right-hand side vector f to the Neumann boundary condition (multiplication of boundary values by 0.5 as well as incorporation of the boundary function g) and/or the Dirichlet boundary condition (setting boundary values to 0 as well as incorporation of the boundary function G). • For the Helmholtz solver on a sphere, the routine omits computation of the spherical weights for the <i>dpar</i>/<i>spar</i> array. • If a=9, the routine omits the normalization of the right-hand side vector f. Depending on the solver, the normalization means: <ul style="list-style-type: none"> • 2D Cartesian case: multiplication by h_y^2, where h_y is the mesh size in the y direction (for details, see Poisson Solver Implementation). • 3D (Cartesian) case: multiplication by h_z^2, where h_z is the mesh size in the z direction. • Helmholtz solver on a sphere: multiplication by h_θ^2, where h_θ is the mesh size in the θ direction (for details, see Poisson Solver Implementation). 	
<p>Using <i>ipar[0]</i> you can adjust the routine to your needs and improve efficiency in solving multiple Helmholtz problems that differ only in the right-hand side. You must be cautious when using this method, because any misunderstanding of the commit process may cause incorrect results or program failure (see also Caveat on Parameter Modifications).</p>	
1	Contains error messaging options:
<ul style="list-style-type: none"> • <i>ipar[1]</i>=-1 indicates that all error messages are printed to the <code>MKL_Poisson_Library_log.txt</code> file in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device (usually, screen). • <i>ipar[1]</i>=0 indicates that no error messages will be printed. • <i>ipar[1]</i>=1 is the default value. It indicates that all error messages are printed to the standard output device. 	
<p>In case of errors, the <i>stat</i> parameter contains a non-zero value on exit from a routine regardless of the <i>ipar[1]</i> setting.</p>	
2	Contains warning messaging options:
<ul style="list-style-type: none"> • <i>ipar[2]</i>=-1 indicates that all warning messages are printed to the <code>MKL_Poisson_Library_log.txt</code> file in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device. • <i>ipar[2]</i>=0 indicates that no warning messages will be printed. • <i>ipar[2]</i>=1 is the default value. It indicates that all warning messages are printed to the standard output device. 	
<p>In case of warnings, the <i>stat</i> parameter contains a non-zero value on exit from a routine regardless of the <i>ipar[2]</i> setting.</p>	
3	Unused.
<p>Parameters 4 through 9 are used only in the Cartesian case.</p>	
4	Takes this value:
<ul style="list-style-type: none"> • 2, if <i>BCType[0]</i>='P' • 1, if <i>BCType[0]</i>='N' 	

Index	Description
	<ul style="list-style-type: none"> • 0, if <code>BCType[0] = 'D'</code> • -1, otherwise
5	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if <code>BCType[1] = 'P'</code> • 1, if <code>BCType[1] = 'N'</code> • 0, if <code>BCType[1] = 'D'</code> • -1, otherwise
6	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if <code>BCType[2] = 'P'</code> • 1, if <code>BCType[2] = 'N'</code> • 0, if <code>BCType[2] = 'D'</code> • -1, otherwise
7	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if <code>BCType[3] = 'P'</code> • 1, if <code>BCType[3] = 'N'</code> • 0, if <code>BCType[3] = 'D'</code> • -1, otherwise
8	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if <code>BCType[4] = 'P'</code> • 1, if <code>BCType[4] = 'N'</code> • 0, if <code>BCType[4] = 'D'</code> • -1, otherwise
9	<p>Takes this value:</p> <ul style="list-style-type: none"> • 2, if <code>BCType[5] = 'P'</code> • 1, if <code>BCType[5] = 'N'</code> • 0, if <code>BCType[5] = 'D'</code> • -1, otherwise
10	<p>Takes the value of</p> <ul style="list-style-type: none"> • nx, that is, the number of intervals along the x-axis, in the Cartesian case. • np, that is, the number of intervals along the φ-axis, in the spherical case.
11	<p>Takes the value of</p> <ul style="list-style-type: none"> • ny, that is, the number of intervals along the y-axis, in the Cartesian case • nt, that is, the number of intervals along the θ-axis, in the spherical case.
12	<p>Takes the value of nz, the number of intervals along the z-axis. This parameter is used only in the 3D case (Cartesian).</p>
13	<p>Has the value of 6, which specifies the internal partitioning of the <code>dpar/spar</code> array.</p>
14	<p>Takes the value of $ipar[13]+ipar[10]+1$, which specifies the internal partitioning of the <code>dpar/spar</code> array.</p>

Index		Description		Cartesian Solver	
				2D case	3D case
15	Unused				Takes the value of $ipar[14]+1$, which specifies the internal partitioning of the $dpar/spar$ array.
16	Unused				Takes the value of $ipar[14]+ipar[11]+1$, which specifies the internal partitioning of the $dpar/spar$ array.
17	Takes the value of $ipar[14]+1$, which specifies the internal partitioning of the $dpar/spar$ array.				Takes the value of $ipar[16]+1$, which specifies the internal partitioning of the $dpar/spar$ array.
18	Takes the value of $ipar[14]+3*ipar[10]/2+1$, which specifies the internal partitioning of the $dpar/spar$ array.		Takes the value of $ipar[16]+3*ipar[10]/2+1$, which specifies the internal partitioning of the $dpar/spar$ array.	Takes the value of $ipar[16]+3*ipar[10]/4+1$, which specifies the internal partitioning of the $dpar/spar$ array.	Takes the value of $ipar[16]+3*ipar[10]/2+1$, which specifies the internal partitioning of the $dpar/spar$ array.
19	Takes the value of $ipar[18]+1$, which specifies the internal partitioning of the $dpar/spar$ array.			Takes the value of $ipar[18]+1$, which specifies the internal partitioning of the $dpar/spar$ array.	Unused
20	Takes the value of $ipar[19]+3*ipar[12]/4$, which specifies the internal partitioning of the $dpar/spar$ array.		Takes the value of $ipar[18]+3*ipar[11]/2+1$, which specifies the internal partitioning of the $dpar/spar$ array.	Takes the value of $ipar[18]+3*ipar[10]/4+1$, which specifies the internal partitioning of the $dpar/spar$ array.	Unused
21	Contains message style options:				
		<ul style="list-style-type: none"> • $ipar[21]=0$ indicates that Poisson Solver routines print all error and warning messages in Fortran-style notations. • $ipar[21]=1$ (default) indicates that Poisson Solver routines print the messages in C-style notations. 			
22	Contains the number of threads to be used for computations in a multithreaded environment. The default value is 1 in the serial mode, and the result returned by the <code>mkl_get_max_threads</code> function otherwise.				
23	Takes the value of $ipar[18]+1$, which specifies the internal partitioning of the $dpar/spar$ array in the periodic Cartesian case.				
24	Takes the value of $ipar[23]+3*ipar[12]/4$, which specifies the internal partitioning of the $dpar/spar$ array in the periodic Cartesian case.				

Index	Description
25	Takes the value of <code>ipar[20]+1</code> , which specifies the internal partitioning of the <code>dpar/spar</code> array in the periodic 3D Cartesian case.
26	Takes the value of <code>ipar[25]+3*ipar[13]/4</code> , which specifies the internal partitioning of the <code>dpar/spar</code> array in the periodic 3D Cartesian case.
27 through 39	Unused.
40 through 59	Contain the first twenty elements of the <code>ipar</code> array of the first Trigonometric Transform that the solver uses. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)
60 through 79	Contain the first twenty elements of the <code>ipar</code> array of the second Trigonometric Transform that the 3D Cartesian and periodic spherical solvers use. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)
80 through 99	Contain the first twenty elements of the <code>ipar</code> array of the third Trigonometric Transform that the solver uses in case of periodic boundary conditions along the <code>x</code> -axis. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)
100 through 119	Contain the first twenty elements of the <code>ipar</code> array of the fourth Trigonometric Transform used by periodic spherical solvers and 3D Cartesian solvers with periodic boundary conditions along the <code>y</code> -axis. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.)

NOTE

While you can declare the `ipar` array as `MKL_INT ipar[120]`, for future compatibility you should declare `ipar` as `MKL_INT ipar[128]`.

dpar and spar

Arrays `dpar` and `spar` are the same except in the data precision:

<code>dpar</code>	Holds data needed for double-precision Fast Helmholtz Solver computations.
	<ul style="list-style-type: none"> For the Cartesian solver, double array of size $5*nx/2+7$ in the 2D case or $5*(nx+ny)/2+9$ in the 3D case; initialized in the <code>d_init_Helmholtz_2D/d_init_Helmholtz_3D</code> and <code>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</code> routines. For the spherical solver, double array of size $5*np/2+nt+10$; initialized in the <code>d_init_sph_p/d_init_sph_np</code> and <code>d_commit_sph_p/d_commit_sph_np</code> routines.
<code>spar</code>	Holds data needed for single-precision Fast Helmholtz Solver computations.
	<ul style="list-style-type: none"> For the Cartesian solver, float array of size $5*nx/2+7$ in the 2D case or $5*(nx+ny)/2+9$ in the 3D case; initialized in the <code>s_init_Helmholtz_2D/s_init_Helmholtz_3D</code> and <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code> routines. For the spherical solver, float array of size $5*np/2+nt+10$; initialized in the <code>s_init_sph_p/s_init_sph_np</code> and <code>s_commit_sph_p/s_commit_sph_np</code> routines.

Because `dpar` and `spar` have similar elements in each position, the elements are described together in [Table "Elements of the dpar and spar Arrays"](#):

Elements of the dpar and spar Arrays

Index	Description
0	<p>In the Cartesian case, contains the length of the interval along the x-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_x in the x direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine.</p> <p>In the spherical case, contains the length of the interval along the ϕ-axis right after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine or the mesh size h_ϕ in the ϕ direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.</p>
1	<p>In the Cartesian case, contains the length of the interval along the y-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_y in the y direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine.</p> <p>In the spherical case, contains the length of the interval along the θ-axis right after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine or the mesh size h_θ in the θ direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.</p>
2	<p>In the Cartesian case, contains the length of the interval along the z-axis right after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D</code> routine or the mesh size h_z in the z direction (for details, see Poisson Solver Implementation) after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine. In the Cartesian solver, this parameter is used only in the 3D case.</p> <p>In the spherical solver, contains the coordinate of the leftmost boundary along the θ-axis after a call to the <code>?_init_sph_p/?_init_sph_np</code> routine.</p>
3	Contains the value of the coefficient q after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np</code> routine.
4	<p>Contains the tolerance parameter after a call to the <code>?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np</code> routine.</p> <ul style="list-style-type: none"> • In the Cartesian case, this value is used only for the pure Neumann boundary conditions (<code>BCtype="NNNN"</code> in the 2D case; <code>BCtype="NNNNNN"</code> in the 3D case). This is a special case, because the right-hand side of the problem cannot be arbitrary if the coefficient q is zero. The Poisson Solver verifies that the classical solution exists (up to rounding errors) using this tolerance. In any case, the Poisson Solver computes the normal solution, that is, the solution that has the minimal Euclidean norm of residual. Nevertheless, the <code>?_Helmholtz_2D/?_Helmholtz_3D</code> routine informs you that the solution may not exist in a classical sense (up to rounding errors). • In the spherical case, the value is used for the special case of a periodic problem on the entire sphere. This special case is similar to the Cartesian case with pure Neumann boundary conditions. Here the Poisson Solver computes the normal solution as well. The parameter is also used to detect whether the problem is periodic up to rounding errors. <p>The default value for this parameter is 1.0E-10 in case of double-precision computations or 1.0E-4 in case of single-precision computations. You can increase the value of the tolerance, for instance, to avoid the warnings that may appear.</p>

Index	Description
<i>ipar</i> [13]-1 through <i>ipar</i> [14]-1	In the Cartesian case, contain the spectrum of the one-dimensional (1D) problem along the x -axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine.
<i>ipar</i> [15]-1 through <i>ipar</i> [16]-1	In the spherical case, contains the spectrum of the 1D problem along the ϕ -axis after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.
<i>ipar</i> [17]-1 through <i>ipar</i> [18]-1	In the Cartesian case, contain the spectrum of the 1D problem along the y -axis after a call to the <code>?_commit_Helmholtz_3D</code> routine. These elements are used only in the 3D case.
<i>ipar</i> [19]-1 through <i>ipar</i> [20]-1	In the spherical case, contains the spherical weights after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine.
<i>ipar</i> [21]-1 through <i>ipar</i> [22]-1	Take the values of the (staggered) sine/cosine in the mesh points:
<ul style="list-style-type: none"> • along the x-axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine for a Cartesian solver • along the ϕ-axis after a call to the <code>?_commit_sph_p/?_commit_sph_np</code> routine for a spherical solver. 	
<i>ipar</i> [23]-1 through <i>ipar</i> [24]-1	Take the values of the (staggered) sine/cosine in the mesh points:
<ul style="list-style-type: none"> • along the y-axis after a call to the <code>?_commit_Helmholtz_3D</code> routine for a Cartesian 3D solver • along the ϕ-axis after a call to the <code>?_commit_sph_p</code> routine for a spherical periodic solver. 	
<i>ipar</i> [25]-1 through <i>ipar</i> [26]-1	These elements are not used in the 2D Cartesian case and in the non-periodic spherical case.
<i>ipar</i> [27]-1 through <i>ipar</i> [28]-1	Take the values of the (staggered) sine/cosine in the mesh points along the x -axis after a call to the <code>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D</code> routine. These elements are used only in the periodic Cartesian case.
<i>ipar</i> [29]-1 through <i>ipar</i> [30]-1	Take the values of the (staggered) sine/cosine in the mesh points along the x -axis after a call to the <code>?_commit_Helmholtz_3D</code> routine. These elements are used only in the periodic 3D Cartesian case.

NOTE

You may define the array size depending upon the type of the problem to solve.

Caveat on Parameter Modifications

Flexibility of the Poisson Solver interface enables you to skip calls to the `?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np` routine and to initialize the basic data structures explicitly in your code. You may also need to modify contents of the *ipar*, *dpar*, and *spar* arrays after initialization. When doing so, provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or incorrect results. You can perform a basic check for correctness and consistency of parameters by calling the `?_commit_Helmholtz_2D/?_commit_Helmholtz_3D` routine; however, this does not ensure the correct solution but only reduces the chance of errors or wrong results.

NOTE

To supply correct and consistent parameters to Poisson Solver routines, you should have considerable experience in using the Poisson Solver interface and good understanding of the solution process, as well as elements contained in the *ipar*, *spar*, and *dpar* arrays and dependencies between values of these elements.

In rare occurrences when you fail in tuning parameters for the Fast Helmholtz Solver, refer for technical support at <http://www.intel.com/software/products/support/>.

WARNING

The only way that ensures a proper solution of a Helmholtz problem is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of *ipar*, *dpar*, and *spar* arrays unless it is necessary.

Parameters That Define Boundary Conditions

Poisson Solver routines for the Cartesian solver use the following common parameters to define the boundary conditions.

Parameters to Define Boundary Conditions for the Cartesian Solver

Parameter	Description
<i>bd_ax</i>	<p>double* for <i>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</i> and <i>d_Helmholtz_2D/d_Helmholtz_3D</i>,</p> <p>float* for <i>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</i> and <i>s_Helmholtz_2D/s_Helmholtz_3D</i>.</p> <p>Contains values of the boundary condition on the leftmost boundary of the domain along the <i>x</i>-axis.</p> <ul style="list-style-type: none"> 2D problem: the size of the array is <i>ny</i>+1. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of <i>BCtype</i>[0] is 'D'): values of the function $G(ax, y_j), j=0, \dots, ny$. Neumann boundary condition (value of <i>BCtype</i>[0] is 'N'): values of the function $g(ax, y_j), j=0, \dots, ny$. The value corresponding to the index <i>j</i> is placed in <i>bd_ax[j]</i>. 3D problem: the size of the array is $(ny+1)*(nz+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of <i>BCtype</i>[0] is 'D'): values of the function $G(ax, y_j, z_k), j=0, \dots, ny, k=0, \dots, nz$. Neumann boundary condition (value of <i>BCtype</i>[0] is 'N'): the values of the function $g(ax, y_j, z_k), j=0, \dots, ny, k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (j, k) is placed in <i>bd_ax[j+k*(ny+1)]</i>. <p>For periodic boundary conditions (the value of <i>BCtype</i>[0] is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<i>bd_bx</i>	<p>double* for <i>d_commit_Helmholtz_2D/d_commit_Helmholtz_3D</i> and <i>d_Helmholtz_2D/d_Helmholtz_3D</i>,</p> <p>float* for <i>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</i> and <i>s_Helmholtz_2D/s_Helmholtz_3D</i>.</p>

Parameter	Description
	<p>Contains values of the boundary condition on the rightmost boundary of the domain along the x-axis.</p> <ul style="list-style-type: none"> 2D problem: the size of the array is $ny+1$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[1]$ is 'D'): values of the function $G(bx, y_j), j=0, \dots, ny$. Neumann boundary condition (value of $BCtype[1]$ is 'N'): values of the function $g(bx, y_j), j=0, \dots, ny$. The value corresponding to the index j is placed in $bd_bx[j]$. 3D problem: the size of the array is $(ny+1)*(nz+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[1]$ is 'D'): values of the function $G(bx, y_j, z_k), j=0, \dots, ny, k=0, \dots, nz$. Neumann boundary condition (value of $BCtype[1]$ is 'N'): values of the function $g(bx, y_j, z_k), j=0, \dots, ny, k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (j, k) is placed in $bd_bx[j+k*(ny+1)]$. <p>For periodic boundary conditions (the value of $BCtype[1]$ is 'P'), this parameter is not used, so it can accept a dummy pointer.</p> <pre>double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D and d_Helmholtz_2D/d_Helmholtz_3D, float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D and s_Helmholtz_2D/s_Helmholtz_3D.</pre> <p>Contains values of the boundary condition on the leftmost boundary of the domain along the y-axis.</p> <ul style="list-style-type: none"> 2D problem: the size of the array is $nx+1$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[2]$ is 'D'): values of the function $G(x_i, ay), i=0, \dots, nx$. Neumann boundary condition (value of $BCtype[2]$ is 'N'): values of the function $g(x_i, ay), i=0, \dots, nx$. The value corresponding to the index i is placed in $bd_ay[i]$. 3D problem: the size of the array is $(nx+1)*(nz+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of $BCtype[2]$ is 'D'): values of the function $G(x_i, ay, z_k), i=0, \dots, nx, k=0, \dots, nz$. Neumann boundary condition (value of $BCtype[2]$ is 'N'): values of the function $g(x_i, ay, z_k), i=0, \dots, nx, k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (i, k) is placed in $bd_ay[i+k*(nx+1)]$. <p>For periodic boundary conditions (the value of $BCtype[2]$ is 'P'), this parameter is not used, so it can accept a dummy pointer.</p> <pre>double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D and d_Helmholtz_2D/d_Helmholtz_3D,</pre>
bd_by	

Parameter	Description
<code>bd_by</code>	<p>float* for <code>s_commit_Helmholtz_2D/s_commit_Helmholtz_3D</code> and <code>s_Helmholtz_2D/s_Helmholtz_3D</code>. Contains values of the boundary condition on the rightmost boundary of the domain along the y-axis.</p> <ul style="list-style-type: none"> 2D problem: the size of the array is $nx+1$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of <code>BCtype[3]</code> is 'D'): values of the function $G(x_i, by), i=0, \dots, nx$. Neumann boundary condition (value of <code>BCtype[3]</code> is 'N'): values of the function $g(x_i, by), i=0, \dots, nx$. The value corresponding to the index i is placed in <code>bd_by[i]</code>. 3D problem: the size of the array is $(nx+1)*(nz+1)$. Its contents depend on the boundary conditions as follows: <ul style="list-style-type: none"> Dirichlet boundary condition (value of <code>BCtype[3]</code> is 'D'): values of the function $G(x_i, by, z_k), i=0, \dots, nx, k=0, \dots, nz$. Neumann boundary condition (value of <code>BCtype[3]</code> is 'N'): values of the function $g(x_i, by, z_k), i=0, \dots, nx, k=0, \dots, nz$. The values are packed in the array so that the value corresponding to indices (i, k) is placed in <code>bd_by[i+k*(nx+1)]</code>. <p>For periodic boundary conditions (the value of <code>BCtype[3]</code> is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<code>bd_az</code>	<pre>double* for <code>d_commit_Helmholtz_3D</code> and <code>d_Helmholtz_3D</code>, float* for <code>s_commit_Helmholtz_3D</code> and <code>s_Helmholtz_3D</code>.</pre> <p>Used only by <code>?_commit_Helmholtz_3D</code> and <code>?_Helmholtz_3D</code>. Contains values of the boundary condition on the leftmost boundary of the domain along the z-axis.</p> <p>The size of the array is $(nx+1)*(ny+1)$. Its contents depend on the boundary conditions as follows:</p> <ul style="list-style-type: none"> Dirichlet boundary condition (value of <code>BCtype[4]</code> is 'D'): values of the function $G(x_i, y_j, az), i=0, \dots, nx, j=0, \dots, ny$. Neumann boundary condition (value of <code>BCtype[4]</code> is 'N'), values of the function $g(x_i, y_j, az), i=0, \dots, nx, j=0, \dots, ny$. <p>The values are packed in the array so that the value corresponding to indices (i, j) is placed in <code>bd_az[i+j*(nx+1)]</code>.</p> <p>For periodic boundary conditions (the value of <code>BCtype[4]</code> is 'P'), this parameter is not used, so it can accept a dummy pointer.</p>
<code>bd_bz</code>	<pre>double* for <code>d_commit_Helmholtz_3D</code> and <code>d_Helmholtz_3D</code>, float* for <code>s_commit_Helmholtz_3D</code> and <code>s_Helmholtz_3D</code>.</pre> <p>Used only by <code>?_commit_Helmholtz_3D</code> and <code>?_Helmholtz_3D</code>. Contains values of the boundary condition on the rightmost boundary of the domain along the z-axis.</p> <p>The size of the array is $(nx+1)*(ny+1)$. Its contents depend on the boundary conditions as follows:</p> <ul style="list-style-type: none"> Dirichlet boundary condition (value of <code>BCtype[5]</code> is 'D'): values of the function $G(x_i, y_j, bz), i=0, \dots, nx, j=0, \dots, ny$. Neumann boundary condition (value of <code>BCtype[5]</code> is 'N'): values of the function $g(x_i, y_j, bz), i=0, \dots, nx, j=0, \dots, ny$.

Parameter	Description
	The values are packed in the array so that the value corresponding to indices (i, j) is placed in $bd_bz[i+j*(nx+1)]$.
	For periodic boundary conditions (the value of $BCtype[5]$ is 'P'), this parameter is not used, so it can accept a dummy pointer.

See Also

[_commit_Helmholtz_2D/?_commit_Helmholtz_3D](#)
[?_Helmholtz_2D/?_Helmholtz_3D](#)

Implementation Details

Several aspects of the Intel MKL Poisson Solver interface are platform-specific and language-specific. To promote portability of the Intel MKL Poisson Solver interface across platforms and ease of use across different languages, Intel MKL provides you with the Poisson Solver language-specific header files to include in your code:

- `mkl_poisson.h`, to be used together with `mkl_dfti.h`, for C programs.
- `mkl_poisson.f90`, to be used together with `mkl_dfti.f90`, for Fortran programs.

NOTE

- Intel MKL Poisson Solver interface supports Fortran versions starting with Fortran 90.
- Use of the Intel MKL Poisson Solver software without including the above language-specific header files is not supported.

C-specific Header File

The C-specific header file defines the following function prototypes for the Cartesian solver:

```
void d_init_Helmholtz_2D(double*, double*, double*, double*, MKL_INT*, MKL_INT*, char*, double*, MKL_INT*, double*, MKL_INT*);
void d_commit_Helmholtz_2D(double*, double*, double*, double*, double*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, double*, MKL_INT*);
void d_Helmholtz_2D(double*, double*, double*, double*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, double*, MKL_INT*);
void s_init_Helmholtz_2D(float*, float*, float*, float*, MKL_INT*, MKL_INT*, char*, float*, MKL_INT*, float*, MKL_INT*);
void s_commit_Helmholtz_2D(float*, float*, float*, float*, float*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, float*, MKL_INT*);
void s_Helmholtz_2D(float*, float*, float*, float*, float*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, float*, MKL_INT*);

void free_Helmholtz_2D(DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, MKL_INT*);

void d_init_Helmholtz_3D(double*, double*, double*, double*, double*, MKL_INT*, MKL_INT*, MKL_INT*, char*, double*, MKL_INT*, double*, MKL_INT*);
void d_commit_Helmholtz_3D(double*, double*, double*, double*, double*, double*, double*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, double*, MKL_INT*);
void d_Helmholtz_3D(double*, double*, double*, double*, double*, double*, double*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, double*, MKL_INT*);

void s_init_Helmholtz_3D(float*, float*, float*, float*, float*, float*, MKL_INT*, MKL_INT*, MKL_INT*, char*, float*, MKL_INT*, float*, MKL_INT*);
void s_commit_Helmholtz_3D(float*, float*, float*, float*, float*, float*, float*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, float*, MKL_INT*);
```

```
void s_Helmholtz_3D(float*, float*, float*, float*, float*, float*, DFTI_DESCRIPTOR_HANDLE*,  
DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, float*, MKL_INT*);  
void free_Helmholtz_3D(DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, MKL_INT*);
```

The C-specific header file defines the following function prototypes for the spherical solver:

```
void d_init_sph_p(double*, double*, double*, double*, MKL_INT*, MKL_INT*, double*, MKL_INT*,  
MKL_INT*);  
void d_commit_sph_p(double*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, double*,  
MKL_INT*);  
void d_sph_p(double*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, double*, MKL_INT*);  
void s_init_sph_p(float*, float*, float*, float*, MKL_INT*, MKL_INT*, float*, MKL_INT*, float*,  
MKL_INT*);  
void s_commit_sph_p(float*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, float*,  
MKL_INT*);  
void s_sph_p(float*, DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, float*, MKL_INT*);  
void free_sph_p(DFTI_DESCRIPTOR_HANDLE*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, MKL_INT*);  
  
void d_init_sph_np(double*, double*, double*, double*, MKL_INT*, MKL_INT*, double*, MKL_INT*,  
MKL_INT*);  
void d_commit_sph_np(double*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, double*, MKL_INT*);  
void d_sph_np(double*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, double*, MKL_INT*);  
void s_init_sph_np(float*, float*, float*, float*, MKL_INT*, MKL_INT*, float*, MKL_INT*, float*,  
MKL_INT*);  
void s_commit_sph_np(float*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, float*, MKL_INT*);  
void s_sph_np(float*, DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, float*, MKL_INT*);  
void free_sph_np(DFTI_DESCRIPTOR_HANDLE*, MKL_INT*, MKL_INT*);
```

Fortran-Specific Header File

The Fortran-Specific header file defines the following function prototypes for the Cartesian solver:

```
SUBROUTINE D_INIT_HELMHOLTZ_2D (AX, BX, AY, BY, NX, NY, BCTYPE, Q, IPAR, DPAR, STAT)  
  USE MKL_DFTI  
  
  INTEGER NX, NY, STAT  
  INTEGER IPAR(*)  
  DOUBLE PRECISION AX, BX, AY, BY, Q  
  DOUBLE PRECISION DPAR(*)  
  CHARACTER(4) BCTYPE  
END SUBROUTINE  
  
  
SUBROUTINE D_COMMIT_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, DPAR, STAT)  
  USE MKL_DFTI  
  
  INTEGER STAT  
  INTEGER IPAR(*)  
  DOUBLE PRECISION F(IPAR(11)+1,:*)  
  DOUBLE PRECISION DPAR(*)  
  DOUBLE PRECISION BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)  
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE  
END SUBROUTINE  
  
  
SUBROUTINE D_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, DPAR, STAT)  
  USE MKL_DFTI  
  
  INTEGER STAT  
  INTEGER IPAR(*)  
  DOUBLE PRECISION F(IPAR(11)+1,:*)  
  DOUBLE PRECISION DPAR(*)
```

```

DOUBLE PRECISION BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE S_INIT_HELMHOLTZ_2D (AX, BX, AY, BY, NX, NY, BCTYPE, Q, IPAR, SPAR, STAT)
USE MKL_DFTI

INTEGER NX, NY, STAT
INTEGER IPAR(*)
REAL AX, BX, AY, BY, Q
REAL SPAR(*)
CHARACTER(4) BCTYPE
END SUBROUTINE

SUBROUTINE S_COMMIT_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, SPAR, STAT)
USE MKL_DFTI

INTEGER STAT
INTEGER IPAR(*)
REAL F(IPAR(11)+1,*)
REAL SPAR(*)
REAL BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE S_HELMHOLTZ_2D (F, BD_AX, BD_BX, BD_AY, BD_BY, XHANDLE, IPAR, SPAR, STAT)
USE MKL_DFTI

INTEGER STAT
INTEGER IPAR(*)
REAL F(IPAR(11)+1,*)
REAL SPAR(*)
REAL BD_AX(*), BD_BX(*), BD_AY(*), BD_BY(*)
TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE FREE_HELMHOLTZ_2D (XHANDLE, IPAR, STAT)
USE MKL_DFTI

INTEGER STAT
INTEGER IPAR(*)
TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE
END SUBROUTINE

SUBROUTINE D_INIT_HELMHOLTZ_3D (AX, BX, AY, BY, AZ, BZ, NX, NY, NZ, BCTYPE, Q, IPAR, DPAR, STAT)
USE MKL_DFTI

INTEGER NX, NY, NZ, STAT
INTEGER IPAR(*)
DOUBLE PRECISION AX, BX, AY, BY, AZ, BZ, Q
DOUBLE PRECISION DPAR(*)
CHARACTER(6) BCTYPE
END SUBROUTINE

```

```

SUBROUTINE D_COMMIT_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE, IPAR,
DPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION F(IPAR(11)+1,IPAR(12)+1,*)
  DOUBLE PRECISION DPAR(*)
  DOUBLE PRECISION BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
  DOUBLE PRECISION BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

SUBROUTINE D_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE, IPAR, DPAR,
STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION F(IPAR(11)+1,IPAR(12)+1,*)
  DOUBLE PRECISION DPAR(*)
  DOUBLE PRECISION BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
  DOUBLE PRECISION BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

SUBROUTINE S_INIT_HELMHOLTZ_3D (AX, BX, AY, BY, AZ, BZ, NX, NY, NZ, BCTYPE, Q, IPAR, SPAR, STAT)
  USE MKL_DFTI

  INTEGER NX, NY, NZ, STAT
  INTEGER IPAR(*)
  REAL AX, BX, AY, BY, AZ, BZ, Q
  REAL SPAR(*)
  CHARACTER(6) BCTYPE
END SUBROUTINE

SUBROUTINE S_COMMIT_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE, IPAR,
SPAR, STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  REAL F(IPAR(11)+1,IPAR(12)+1,*)
  REAL SPAR(*)
  REAL BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
  REAL BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

SUBROUTINE S_HELMHOLTZ_3D (F, BD_AX, BD_BX, BD_AY, BD_BY, BD_AZ, BD_BZ, XHANDLE, YHANDLE, IPAR, SPAR,
STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)

```

```

REAL F(IPAR(11)+1,IPAR(12)+1,*)
REAL SPAR(*)
REAL BD_AX(IPAR(12)+1,*), BD_BX(IPAR(12)+1,*), BD_AY(IPAR(11)+1,*)
REAL BD_BY(IPAR(11)+1,*), BD_AZ(IPAR(11)+1,*), BD_BZ(IPAR(11)+1,*)
TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

SUBROUTINE FREE_HELMHOLTZ_3D (XHANDLE, YHANDLE, IPAR, STAT)
USE MKL_DFTI

INTEGER STAT
INTEGER IPAR(*)
TYPE(DFTI_DESCRIPTOR), POINTER :: XHANDLE, YHANDLE
END SUBROUTINE

```

The Fortran-Specific header file defines the following function prototypes for the spherical solver:

```

SUBROUTINE D_INIT_SPH_P(AP,BP,AT,BT,NP,NT,Q,IPAR,DPAR,STAT)
USE MKL_DFTI

INTEGER NP, NT, STAT
INTEGER IPAR(*)
DOUBLE PRECISION AP,BP,AT,BT,Q
DOUBLE PRECISION DPAR(*)
END SUBROUTINE

SUBROUTINE D_COMMIT_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,DPAR,STAT)
USE MKL_DFTI

INTEGER STAT
INTEGER IPAR(*)
DOUBLE PRECISION DPAR(*)
DOUBLE PRECISION F(IPAR(11)+1,*)
TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

SUBROUTINE D_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,DPAR,STAT)
USE MKL_DFTI

INTEGER STAT
INTEGER IPAR(*)
DOUBLE PRECISION DPAR(*)
DOUBLE PRECISION F(IPAR(11)+1,*)
TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE

SUBROUTINE S_INIT_SPH_P(AP,BP,AT,BT,NP,NT,Q,IPAR,SPAR,STAT)
USE MKL_DFTI

INTEGER NP, NT, STAT
INTEGER IPAR(*)
REAL AP,BP,AT,BT,Q
REAL SPAR(*)
END SUBROUTINE

```

```
SUBROUTINE S_COMMIT_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,SPAR,STAT)
  USE MKL_DFTI
```

```
  INTEGER STAT
  INTEGER IPAR(*)
  REAL SPAR(*)
  REAL F(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE
```

```
SUBROUTINE S_SPH_P(F,HANDLE_S,HANDLE_C,IPAR,SPAR,STAT)
  USE MKL_DFTI
```

```
  INTEGER STAT
  INTEGER IPAR(*)
  REAL SPAR(*)
  REAL F(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_C, HANDLE_S
END SUBROUTINE
```

```
SUBROUTINE FREE_SPH_P(HANDLE_S,HANDLE_C,IPAR,STAT)
  USE MKL_DFTI
```

```
  INTEGER STAT
  INTEGER IPAR(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE_S, HANDLE_C
END SUBROUTINE
```

```
SUBROUTINE D_INIT_SPH_NP(AP,BP,AT,BT,NP,NT,Q,IPAR,DPAR,STAT)
  USE MKL_DFTI
```

```
  INTEGER NP, NT, STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION AP,BP,AT,BT,Q
  DOUBLE PRECISION DPAR(*)
END SUBROUTINE
```

```
SUBROUTINE D_COMMIT_SPH_NP(F,HANDLE,IPAR,DPAR,STAT)
  USE MKL_DFTI
```

```
  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION DPAR(*)
  DOUBLE PRECISION F(IPAR(11)+1,*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE
```

```
SUBROUTINE D_SPH_NP(F,HANDLE,IPAR,DPAR,STAT)
  USE MKL_DFTI
```

```
  INTEGER STAT
  INTEGER IPAR(*)
  DOUBLE PRECISION DPAR(*)
  DOUBLE PRECISION F(IPAR(11)+1,*)
```

```

      TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

SUBROUTINE S_INIT_SPH_NP(AP,BP,AT,BT,NP,NT,Q,IPAR,SPAR,STAT)
  USE MKL_DFTI

  INTEGER NP, NT, STAT
  INTEGER IPAR(*)
  REAL AP,BP,AT,BT,Q
  REAL SPAR(*)
END SUBROUTINE

SUBROUTINE S_COMMIT_SPH_NP(F,HANDLE,IPAR,SPAR,STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  REAL SPAR(*)
  REAL F(IPAR(11)+1,:)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

SUBROUTINE S_SPH_NP(F,HANDLE,IPAR,SPAR,STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  REAL SPAR(*)
  REAL F(IPAR(11)+1,:)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

SUBROUTINE FREE_SPH_NP(HANDLE,IPAR,STAT)
  USE MKL_DFTI

  INTEGER STAT
  INTEGER IPAR(*)
  TYPE(DFTI_DESCRIPTOR), POINTER :: HANDLE
END SUBROUTINE

```

Fortran specifics of the Poisson Solver routines usage are similar for all Intel MKL PDE support tools and described in the [Calling PDE Support Routines from Fortran](#) section.

Calling PDE Support Routines from Fortran

The calling interface for all the Intel MKL TT and Poisson Solver routines is designed to be easily used in C. However, you can invoke each TT or Poisson Solver routine directly from Fortran 90 or higher if you are familiar with the inter-language calling conventions of your platform.

The TT or Poisson Solver interface cannot be invoked from FORTRAN 77 due to restrictions imposed by the use of the Intel MKL FFT interface.

The inter-language calling conventions include, but are not limited to, the argument passing mechanisms for the language, the data type mappings from C to Fortran, and how C external names are decorated on the platform.

To promote portability and relieve you of dealing with the calling conventions specifics, the Fortran header file `mkl_trig_transforms.f90` for TT routines and `mkl_poisson.f90` for Poisson Solver routines, used together with `mkl_dfti.f90`, declare a set of macros and introduce type definitions intended to hide the inter-language calling conventions and provide an interface to the routines that looks natural in Fortran.

For example, consider a hypothetical library routine, `foo`, which takes a double-precision vector of length n . C users access such a function as follows:

```
MKL_INT n;
double *x;
...
foo(x, &n);
```

As noted above, to invoke `foo`, Fortran users would need to know what Fortran data types correspond to C types `MKL_INT` and `double` (or `float` for single-precision), what argument-passing mechanism the C compiler uses and what, if any, name decoration is performed by the C compiler when generating the external symbol `foo`. However, with the Fortran header files `mkl_trig_transforms.f90` / `mkl_poisson.f90` and `mkl_dfti.f90` included, the invocation of `foo` within a Fortran program will look as follows for the LP64 interface (for the ILP64 interface, `INTEGER*8` type will be used instead of `INTEGER*4`):

- For TT interface,

```
use mkl_dfti
use mkl_trig_transforms
INTEGER*4 n
DOUBLE PRECISION, ALLOCATABLE :: x
...
CALL FOO(x,n)
```

- For Poisson Solver interface,

```
use mkl_dfti
use mkl_poisson
INTEGER*4 n
DOUBLE PRECISION, ALLOCATABLE :: x
...
CALL FOO(x,n)
```

Note that in the above example, the header files `mkl_trig_transforms.f90` / `mkl_poisson.f90` and `mkl_dfti.f90` provide a definition for the subroutine `FOO`. To ease the use of Poisson Solver or TT routines in Fortran, the general approach of providing Fortran definitions of names is used throughout the libraries. Specifically, if a name from a Poisson Solver or TT interface is documented as having the C-specific name `foo`, then the Fortran header files provide an appropriate Fortran language type definition `FOO`.

One of the key differences between Fortran and C is the language argument-passing mechanism: C programs use pass-by-value semantics and Fortran programs use pass-by-reference semantics. The Fortran headers ensure proper treatment of this difference. In particular, in the above example, the header files `mkl_trig_transforms.f90` / `mkl_poisson.f90` and `mkl_dfti.f90` hide the difference by defining a macro `FOO` that takes the address of the appropriate arguments.

See Also

[C Datatypes Specific to Intel MKL](#)

Nonlinear Optimization Problem Solvers

13

Intel® Math Kernel Library (Intel® MKL) provides tools for solving nonlinear least squares problems using the Trust-Region (TR) algorithms. The solver routines are grouped according to their purpose as follows:

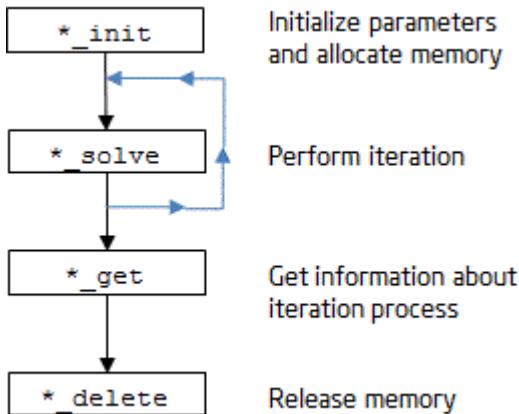
- Nonlinear Least Squares Problem without Constraints
- Nonlinear Least Squares Problem with Linear (Boundary) Constraints
- Jacobian Matrix Calculation Routines

For more information on the key concepts required to understand the use of the Intel MKL nonlinear least squares problem solver routines, see [Conn00].

Organization and Implementation

The Intel MKL solver routines for nonlinear least squares problems use reverse communication interfaces (RCI). That means you need to provide the solver with information required for the iteration process, for example, the corresponding Jacobian matrix, or values of the objective function. RCI removes the dependency of the solver on specific implementation of the operations. However, it does require that you organize a computational loop.

Typical order for invoking RCI solver routines



The nonlinear least squares problem solver routines, or Trust-Region (TR) solvers, are implemented with the OpenMP* support. You can manage the threads using [threading control functions](#). The TR solvers use BLAS and LAPACK routines, and offer the same parallelism as those domains. The ?jacobi and ?jacobix routines of Jacobi matrix calculations are parallel. These routines (?jacobi and ?jacobix) make calls to the user-supplied functions with different x parameters for multiple threads.

Memory Allocation and Handles

To make the TR solver routines easy to use, you are not required to allocate temporary working storage. The solver allocates any required memory. To allow multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using a data object called a *handle*. Each TR solver routine creates, uses, or deletes a handle. To declare a handle, include `mkl_rci.h` or `mkl_rci.fi`.

For a C and C++ program, declare a handle as one of the following:

```
#include "mkl_rci.h"
_TRNSP_HANDLE_t handle;
```

or

```
_TRNSPBC_HANDLE_t handle;
```

The first declaration is used for nonlinear least squares problems without boundary constraints, and the second is used for nonlinear least squares problems with boundary constraints.

For a Fortran program using compilers that support eight byte integers, declare a handle as:

```
INCLUDE "mkl_rci.fi"
INTEGER*8 handle
```

Routine Naming Conventions

The TR routine names have the following structure:

```
<character><name>_<action>( )
```

where

- *<character>* indicates the data type:

s	real, single precision
d	real, double precision

- *<name>* indicates the task type:

trnlsp	nonlinear least squares problem without constraints
trnlspbc	nonlinear least squares problem with boundary constraints
jacobi	computation of the Jacobian matrix using central differences

- *<action>* indicates an action on the task:

init	initializes the solver
check	checks correctness of the input parameters
solve	solves the problem
get	retrieves the number of iterations, the stop criterion, the initial residual, and the final residual
delete	releases the allocated data

Nonlinear Least Squares Problem without Constraints

The nonlinear least squares problem without constraints can be described as follows:

$$\min_{x \in R^n} \|F(x)\|_2^2 = \min_{x \in R^n} \|y - f(x)\|_2^2, y \in R^m, x \in R^n, f: R^n \rightarrow R^m, m \geq n,$$

where

$F(x) : R^n \rightarrow R^m$ is a twice differentiable function in R^n .

Solving a nonlinear least squares problem means searching for the best approximation to the vector y with the model function $f_i(x)$ and nonlinear variables x . The best approximation means that the sum of squares of residuals $y_i - f_i(x)$ is the minimum.

See usage examples in Fortran and C in the `examples\solverf\source` and `examples\solverc\source` folders of your Intel MKL directory, respectively. Specifically, see `ex_nlsqp_f.f` and `ex_nlsqp_c.c`.

RCI TR Routines

Routine Name	Operation
?trnlsp_init	Initializes the solver.
?trnlsp_check	Checks correctness of the input parameters.
?trnlsp_solve	Solves a nonlinear least squares problem using the Trust-Region algorithm.
?trnlsp_get	Retrieves the number of iterations, stop criterion, initial residual, and final residual.
?trnlsp_delete	Releases allocated data.

?trnlsp_init

Initializes the solver of a nonlinear least squares problem.

Syntax

Fortran:

```
res = strnlsp_init(handle, n, m, x, eps, iter1, iter2, rs)
res = dtrnlsp_init(handle, n, m, x, eps, iter1, iter2, rs)
```

C:

```
MKL_INT strnlsp_init (_TRNSP_HANDLE_t* &handle, MKL_INT* &n, MKL_INT* &m, float* x,
float* &eps, MKL_INT* &iter1, MKL_INT* &iter2, float* &rs);
MKL_INT dtrnlsp_init (_TRNSP_HANDLE_t* &handle, MKL_INT* &n, MKL_INT* &m, double* x,
double* &eps, MKL_INT* &iter1, MKL_INT* &iter2, double* &rs);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The `?trnlsp_init` routine initializes the solver.

After initialization, all subsequent invocations of the `?trnlsp_solve` routine should use the values of the handle returned by `?trnlsp_init`.

The `eps` array contains the stopping criteria:

eps Value	Description
1	$\Delta < \text{eps}(1)$
2	$\ F(x)\ _2 < \text{eps}(2)$

eps Value	Description
3	The Jacobian matrix is singular. $\ \mathcal{J}(x)_{(1:m,j)} \ _2 < \text{eps}(3), j = 1, \dots, n$
4	$\ s \ _2 < \text{eps}(4)$
5	$\ F(x) \ _2 - \ F(x) - \mathcal{J}(x)s \ _2 < \text{eps}(5)$
6	The trial step precision. If $\text{eps}(6) = 0$, then the trial step meets the required precision ($\leq 1.0D-10$).

Note:

- $\mathcal{J}(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

<i>n</i>	INTEGER. Length of x .
<i>m</i>	INTEGER. Length of $F(x)$.
<i>x</i>	REAL for strnlsp_init DOUBLE PRECISION for dtrnlsp_init Array of size <i>n</i> . Initial guess.
<i>eps</i>	REAL for strnlsp_init DOUBLE PRECISION for dtrnlsp_init Array of size 6; contains stopping criteria. See the values in the Description section.
<i>iter1</i>	INTEGER. Specifies the maximum number of iterations.
<i>iter2</i>	INTEGER. Specifies the maximum number of iterations of trial step calculation.
<i>rs</i>	REAL for strnlsp_init DOUBLE PRECISION for dtrnlsp_init Definition of initial size of the trust region (boundary of the trial step). The recommend minimum value is 0.1, and the recommended maximum value is 100.0. Based on your knowledge of the objective function and initial guess you can increase or decrease the initial trust region. It can influence the iteration process, for example, the direction of the iteration process and the number of iterations. If you set <i>rs</i> to 0.0, the solver uses the default value, which is 100.0.

Output Parameters

<i>handle</i>	Type _TRNSP_HANDLE_t in C/C++ and INTEGER*8 in FORTRAN.
<i>res</i>	INTEGER. Indicates task completion status.

- `res = TR_SUCCESS` - the routine completed the task normally.
- `res = TR_INVALID_OPTION` - there was an error in the input parameters.
- `res = TR_OUT_OF_MEMORY` - there was a memory error.

`TR_SUCCESS`, `TR_INVALID_OPTION`, and `TR_OUT_OF_MEMORY` are defined in the `mkl_rci.fi` (Fortran) and `mkl_rci.h` (C) include files.

See Also

[?trnlsp_solve](#)

?trnlsp_check

Checks the correctness of handle and arrays containing Jacobian matrix, objective function, and stopping criteria.

Syntax

Fortran:

```
res = strnlsp_check(handle, n, m, fjac, fvec, eps, info)
res = dtrnlsp_check(handle, n, m, fjac, fvec, eps, info)
```

C:

```
MKL_INT strnlsp_check (_TRNSP_HANDLE_t* &handle, MKL_INT* &n, MKL_INT* &m, float* fjac,
float* fvec, float* eps, MKL_INT* info);

MKL_INT dtrnlsp_check (_TRNSP_HANDLE_t* &handle, MKL_INT* &n, MKL_INT* &m, double* fjac,
double* fvec, double* eps, MKL_INT* info);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The `?trnlsp_check` routine checks the arrays passed into the solver as input parameters. If an array contains any `INF` or `Nan` values, the routine sets the flag in output array `info` (see the description of the values returned in the Output Parameters section for the `info` array).

Input Parameters

<code>handle</code>	Type <code>_TRNSPBC_HANDLE_t</code> in C/C++ and <code>INTEGER*8</code> in FORTRAN.
<code>n</code>	<code>INTEGER</code> . Length of x .
<code>m</code>	<code>INTEGER</code> . Length of $F(x)$.
<code>fjac</code>	<code>REAL</code> for <code>strnlsp_check</code> <code>DOUBLE PRECISION</code> for <code>dtrnlsp_check</code> Array of size m by n . Contains the Jacobian matrix of the function.
<code>fvec</code>	<code>REAL</code> for <code>strnlsp_check</code>

DOUBLE PRECISION for `dtrnlsp_check`
 Array of size m . Contains the function values at x , where $fvec(i) = (y_i - f_i(x))$.

`eps` REAL for `strnlsp_check`

DOUBLE PRECISION for `dtrnlsp_check`

Array of size 6; contains stopping criteria. See the values in the Description section of the [?trnlsp_init](#).

Output Parameters

`info` INTEGER

Array of size 6.

Results of input parameter checking:

Parameter	Used for	Value	Description
<code>FORTRAN: info(1)</code> <code>C: info[0]</code>	Flags for handle	0 1	The handle is valid. The handle is not allocated.
<code>FORTRAN: info(2)</code> <code>C: info[1]</code>	Flags for <i>fjac</i>	0 1 2 3	The <i>fjac</i> array is valid. The <i>fjac</i> array is not allocated The <i>fjac</i> array contains NaN. The <i>fjac</i> array contains Inf.
<code>FORTRAN: info(3)</code> <code>C: info[2]</code>	Flags for <i>fvec</i>	0 1 2 3	The <i>fvec</i> array is valid. The <i>fvec</i> array is not allocated The <i>fvec</i> array contains NaN. The <i>fvec</i> array contains Inf.
<code>FORTRAN: info(4)</code> <code>C: info[3]</code>	Flags for <i>eps</i>	0 1 2 3 4	The <i>eps</i> array is valid. The <i>eps</i> array is not allocated The <i>eps</i> array contains NaN. The <i>eps</i> array contains Inf. The <i>eps</i> array contains a value less than or equal to zero.

`res` INTEGER. Information about completion of the task.

`res = TR_SUCCESS` - the routine completed the task normally.

`TR_SUCCESS` is defined in the `mkl_rci.h` and `mkl_rci.fi` include files.

?trnlsp_solve

Solves a nonlinear least squares problem using the TR algorithm.

Syntax

Fortran:

```
res = strnlsp_solve(handle, fvec, fjac, RCI_Request)
res = dtrnlsp_solve(handle, fvec, fjac, RCI_Request)
```

C:

```
MKL_INT strnlsp_solve (_TRNSP_HANDLE_t* &handle, float* fvec, float* fjac, MKL_INT* &RCI_Request);
MKL_INT dtrnlsp_solve (_TRNSP_HANDLE_t* &handle, double* fvec, double* fjac, MKL_INT* &RCI_Request);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The ?trnlsp_solve routine uses the TR algorithm to solve nonlinear least squares problems.

The problem is stated as follows:

$$\min_{x \in \mathbb{R}^n} \|F(x)\|_2^2 = \min_{x \in \mathbb{R}^n} \|y - f(x)\|_2^2, y \in \mathbb{R}^m, x \in \mathbb{R}^n, f: \mathbb{R}^n \rightarrow \mathbb{R}^m, m \geq n,$$

where

- $F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$
- $m \geq n$

From a current point $x_{current}$, the algorithm uses the trust-region approach:

$$\min_{x \in \mathbb{R}^n} \|F(x_{current}) + J(x_{current})(x_{new} - x_{current})\|_2^2 \quad \text{subject to } \|x_{new} - x_{current}\| \leq \Delta_{current}$$

to get $x_{new} = x_{current} + s$ that satisfies

$$\min_{s \in \mathbb{R}^n} \|J^T(x)J(x)s + J^T F(x)\|_2^2$$

where

- $J(x)$ is the Jacobian matrix
- s is the trial step

- $\|s\|_2 \leq \Delta_{current}$

The *RCI_Request* parameter provides additional information:

<i>RCI_Request</i> Value	Description
2	Request to calculate the Jacobian matrix and put the result into <i>fjac</i>
1	Request to recalculate the function at vector <i>x</i> and put the result into <i>fvec</i>
0	One successful iteration step on the current trust-region radius (that does not mean that the value of <i>x</i> has changed)
-1	The algorithm has exceeded the maximum number of iterations
-2	$\Delta < \text{eps}(1)$
-3	$\ F(x)\ _2 < \text{eps}(2)$
-4	The Jacobian matrix is singular. $\ \mathcal{J}(x)_{(1:m, j)}\ _2 < \text{eps}(3), j = 1, \dots, n$
-5	$\ s\ _2 < \text{eps}(4)$
-6	$\ F(x)\ _2 - \ F(x) - \mathcal{J}(x)s\ _2 < \text{eps}(5)$

Note:

- $\mathcal{J}(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

<i>handle</i>	Type <code>_TRNSP_HANDLE_t</code> in C/C++ and <code>INTEGER*8</code> in FORTRAN.
<i>fvec</i>	REAL for <code>strnlsp_solve</code> DOUBLE PRECISION for <code>dtrnlsp_solve</code> Array of size <i>m</i> . Contains the function values at <i>x</i> , where $fvec(i) = (y_i - f_i(x))$.
<i>fjac</i>	REAL for <code>strnlsp_solve</code> DOUBLE PRECISION for <code>dtrnlsp_solve</code> Array of size (m, n) . Contains the Jacobian matrix of the function.

Output Parameters

<i>fvec</i>	REAL for <code>strnlsp_solve</code> DOUBLE PRECISION for <code>dtrnlsp_solve</code> Array of size <i>m</i> . Updated function evaluated at <i>x</i> .
<i>RCI_Request</i>	INTEGER. Informs about the task stage. See the Description section for the parameter values and their meaning.

res INTEGER. Indicates the task completion.
res = TR_SUCCESS - the routine completed the task normally.
 TR_SUCCESS is defined in the `mkl_rci.h` and `mkl_rci.fi` include files.

?trnlsp_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax

Fortran:

```
res = strnlsp_get(handle, iter, st_cr, r1, r2)
res = dtrnlsp_get(handle, iter, st_cr, r1, r2)
```

C:

```
MKL_INT strnlsp_get(_TRNSP_HANDLE_t* &handle, MKL_INT* &iter, MKL_INT* &st_cr, float* &r1, float* &r2);
MKL_INT dtrnlsp_get (_TRNSP_HANDLE_t* &handle, MKL_INT* &iter, MKL_INT* &st_cr, double* &r1, double* &r2);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The routine retrieves the current number of iterations, the stop criterion, the initial residual, and final residual.

The initial residual is the value of the functional $(\|y - f(x)\|)$ of the initial x values provided by the user.

The final residual is the value of the functional $(\|y - f(x)\|)$ of the final x resulting from the algorithm operation.

The *st_cr* parameter contains the stop criterion:

st_cr Value	Description
1	The algorithm has exceeded the maximum number of iterations
2	$\Delta < \text{eps}(1)$
3	$\ F(x)\ _2 < \text{eps}(2)$
4	The Jacobian matrix is singular. $\ \mathcal{J}(x)_{(1:m,j)}\ _2 < \text{eps}(3), j = 1, \dots, n$
5	$\ s\ _2 < \text{eps}(4)$
6	$\ F(x)\ _2 - \ F(x) - \mathcal{J}(x)s\ _2 < \text{eps}(5)$

Note:

- $\mathcal{J}(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

handle Type `_TRNSP_HANDLE_t` in C/C++ and `INTEGER*8` in FORTRAN.

Output Parameters

iter `INTEGER`. Contains the current number of iterations.

st_cr `INTEGER`. Contains the stop criterion.

See the Description section for the parameter values and their meanings.

r1 `REAL` for `strnlsp_get`

`DOUBLE PRECISION` for `dtrnlsp_get`

Contains the residual, $(| | y - f(x) | |)$ given the initial x .

r2 `REAL` for `strnlsp_get`

`DOUBLE PRECISION` for `dtrnlsp_get`

Contains the final residual, that is, the value of the functional $(| | y - f(x) | |)$ of the final x resulting from the algorithm operation.

res `INTEGER`. Indicates the task completion.

`res = TR_SUCCESS` - the routine completed the task normally.

`TR_SUCCESS` is defined in the `mkl_rci.h` and `mkl_rci.fi` include files.

?trnlsp_delete

Releases allocated data.

Syntax

Fortran:

```
res = strnlsp_delete(handle)
res = dtrnlsp_delete(handle)
```

C:

```
MKL_INT strnlsp_delete(_TRNSP_HANDLE_t* &handle);
MKL_INT dtrnlsp_delete(_TRNSP_HANDLE_t* &handle);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The `?trnlsp_delete` routine releases all memory allocated for the handle.

This routine flags memory as not used, but to actually release all memory you must call the support function `mkl_free_buffers`.

Input Parameters

`handle` Type `_TRNSP_HANDLE_t` in C/C++ and `INTEGER*8` in FORTRAN.

Output Parameters

`res` `INTEGER`. Indicates the task completion.

`res = TR_SUCCESS` means the routine completed the task normally.

`TR_SUCCESS` is defined in the `mkl_rci.h` and `mkl_rci.fi` include files.

Nonlinear Least Squares Problem with Linear (Bound) Constraints

The nonlinear least squares problem with linear bound constraints is very similar to the [nonlinear least squares problem without constraints](#) but it has the following constraints:

$$l_i \leq x_i \leq u_i, i = 1, \dots, n, \quad l, u \in R^n.$$

See usage examples in Fortran and C in the `examples\solverf\source` and `examples\solverc\source` folders of your Intel MKL directory, respectively. Specifically, see `ex_nlsqp_bc_f.f` and `ex_nlsqp_bc_c.c`.

RCI TR Routines for Problem with Bound Constraints

Routine Name	Operation
<code>?trnlspbc_init</code>	Initializes the solver.
<code>?trnlspbc_check</code>	Checks correctness of the input parameters.
<code>?trnlspbc_solve</code>	Solves a nonlinear least squares problem using RCI and the Trust-Region algorithm.
<code>?trnlspbc_get</code>	Retrieves the number of iterations, stop criterion, initial residual, and final residual.
<code>?trnlspbc_delete</code>	Releases allocated data.

`?trnlspbc_init`

Initializes the solver of nonlinear least squares problem with linear (boundary) constraints.

Syntax

Fortran:

```
res = strnlspbc_init(handle, n, m, x, LW, UP, eps, iter1, iter2, rs)
res = dtrnlspbc_init(handle, n, m, x, LW, UP, eps, iter1, iter2, rs)
```

C:

```

MKL_INT strnlspbc_init (_TRNSPBC_HANDLE_t* &handle, MKL_INT* &n, MKL_INT* &m, float* x,
float* LW, float* UP, float* eps, MKL_INT* &iter1, MKL_INT* &iter2, float* &rs);

MKL_INT dtrnlspbc_init (_TRNSPBC_HANDLE_t* &handle, MKL_INT* &n, MKL_INT* &m, double* x,
double* LW, double* UP, double* eps, MKL_INT* &iter1, MKL_INT* &iter2, double* &rs);

```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The ?trnlspbc_init routine initializes the solver.

After initialization all subsequent invocations of the ?trnlspbc_solve routine should use the values of the handle returned by ?trnlspbc_init.

The *eps* array contains the stopping criteria:

<i>eps</i> Value	Description
1	$\Delta < \text{eps}(1)$
2	$\ F(x) \ _2 < \text{eps}(2)$
3	The Jacobian matrix is singular. $\ J(x)_{(1:m, j)} \ _2 < \text{eps}(3), j = 1, \dots, n$
4	$\ s \ _2 < \text{eps}(4)$
5	$\ F(x) \ _2 - \ F(x) - J(x)s \ _2 < \text{eps}(5)$
6	The trial step precision. If $\text{eps}(6) = 0$, then the trial step meets the required precision ($\leq 1.0D-10$).

Note:

- $J(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

<i>n</i>	INTEGER. Length of <i>x</i> .
<i>m</i>	INTEGER. Length of $F(x)$.
<i>x</i>	REAL for strnlspbc_init DOUBLE PRECISION for dtrnlspbc_init Array of size <i>n</i> . Initial guess.
<i>LW</i>	REAL for strnlspbc_init DOUBLE PRECISION for dtrnlspbc_init

	Array of size n .
	Contains low bounds for x ($lw_i < x_i$).
<i>UP</i>	REAL for strnlspbc_init DOUBLE PRECISION for dtrnlspbc_init
	Array of size n .
	Contains upper bounds for x ($up_i > x_i$).
<i>eps</i>	REAL for strnlspbc_init DOUBLE PRECISION for dtrnlspbc_init
	Array of size 6; contains stopping criteria. See the values in the Description section.
<i>iter1</i>	INTEGER. Specifies the maximum number of iterations.
<i>iter2</i>	INTEGER. Specifies the maximum number of iterations of trial step calculation.
<i>rs</i>	REAL for strnlspbc_init DOUBLE PRECISION for dtrnlspbc_init Definition of initial size of the trust region (boundary of the trial step). The recommended minimum value is 0.1, and the recommended maximum value is 100.0. Based on your knowledge of the objective function and initial guess you can increase or decrease the initial trust region. It can influence the iteration process, for example, the direction of the iteration process and the number of iterations. If you set <i>rs</i> to 0.0, the solver uses the default value, which is 100.0.

Output Parameters

<i>handle</i>	Type _TRNSPBC_HANDLE_t in C/C++ and INTEGER*8 in FORTRAN.
<i>res</i>	INTEGER. Informs about the task completion. <ul style="list-style-type: none"> • <i>res</i> = TR_SUCCESS - the routine completed the task normally. • <i>res</i> = TR_INVALID_OPTION - there was an error in the input parameters. • <i>res</i> = TR_OUT_OF_MEMORY - there was a memory error. TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.fi (Fortran) and mkl_rci.h (C) include files.

?trnlspbc_check

Checks the correctness of *handle* and arrays containing Jacobian matrix, objective function, lower and upper bounds, and stopping criteria.

Syntax

Fortran:

```
res = strnlspbc_check(handle, n, m, fjac, fvec, LW, UP, eps, info)
res = dtrnlspbc_check(handle, n, m, fjac, fvec, LW, UP, eps, info)
```

C:

```
MKL_INT strnlspbc_check (_TRNSPBC_HANDLE_t* &handle, MKL_INT* &n, MKL_INT* &m, float* fjac, float* fvec, float* LW, float* UP, float* eps, MKL_INT* info);

MKL_INT dtrnlspbc_check (_TRNSPBC_HANDLE_t* &handle, MKL_INT* &n, MKL_INT* &m, double* fjac, double* fvec, double* LW, double* UP, double* eps, MKL_INT* info);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The `?trnlspbc_check` routine checks the arrays passed into the solver as input parameters. If an array contains any `INF` or `NAN` values, the routine sets the flag in output array `info` (see the description of the values returned in the Output Parameters section for the `info` array).

Input Parameters

<code>handle</code>	Type <code>_TRNSPBC_HANDLE_t</code> in C/C++ and <code>INTEGER*8</code> in FORTRAN.
<code>n</code>	INTEGER. Length of x .
<code>m</code>	INTEGER. Length of $F(x)$.
<code>fjac</code>	REAL for <code>strnlspbc_check</code> DOUBLE PRECISION for <code>dtrnlspbc_check</code> Array of size m by n . Contains the Jacobian matrix of the function.
<code>fvec</code>	REAL for <code>strnlspbc_check</code> DOUBLE PRECISION for <code>dtrnlspbc_check</code> Array of size m . Contains the function values at x , where $fvec(i) = (y_i - f_i(x))$.
<code>LW</code>	REAL for <code>strnlspbc_check</code> DOUBLE PRECISION for <code>dtrnlspbc_check</code> Array of size n . Contains low bounds for x ($lw_i < x_i$).
<code>UP</code>	REAL for <code>strnlspbc_check</code> DOUBLE PRECISION for <code>dtrnlspbc_check</code> Array of size n . Contains upper bounds for x ($up_i > x_i$).
<code>eps</code>	REAL for <code>strnlspbc_check</code> DOUBLE PRECISION for <code>dtrnlspbc_check</code> Array of size 6; contains stopping criteria. See the values in the Description section of the <code>?trnlspbc_init</code> .

Output Parameters

info

INTEGER

Array of size 6.

Results of input parameter checking:

Parameter	Used for	Value	Description
FORTTRAN: <i>info</i> (1) C: <i>info</i> [0]	Flags for handle	0 1	The handle is valid. The handle is not allocated.
FORTTRAN: <i>info</i> (2) C: <i>info</i> [1]	Flags for <i>fjac</i>	0 1 2 3	The <i>fjac</i> array is valid. The <i>fjac</i> array is not allocated The <i>fjac</i> array contains NaN. The <i>fjac</i> array contains Inf.
FORTTRAN: <i>info</i> (3) C: <i>info</i> [2]	Flags for <i>fvec</i>	0 1 2 3	The <i>fvec</i> array is valid. The <i>fvec</i> array is not allocated The <i>fvec</i> array contains NaN. The <i>fvec</i> array contains Inf.
FORTTRAN: <i>info</i> (4) C: <i>info</i> [3]	Flags for <i>LW</i>	0 1 2 3 4	The <i>LW</i> array is valid. The <i>LW</i> array is not allocated The <i>LW</i> array contains NaN. The <i>LW</i> array contains Inf. The lower bound is greater than the upper bound.
FORTTRAN: <i>info</i> (5) C: <i>info</i> [4]	Flags for <i>up</i>	0 1 2 3 4	The <i>up</i> array is valid. The <i>up</i> array is not allocated The <i>up</i> array contains NaN. The <i>up</i> array contains Inf. The upper bound is less than the lower bound.
FORTTRAN: <i>info</i> (6) C: <i>info</i> [5]	Flags for <i>eps</i>	0 1 2 3	The <i>eps</i> array is valid. The <i>eps</i> array is not allocated The <i>eps</i> array contains NaN. The <i>eps</i> array contains Inf.

Parameter	Used for	Value	Description
		4	The <i>eps</i> array contains a value less than or equal to zero.

res

INTEGER. Information about completion of the task.

res = TR_SUCCESS - the routine completed the task normally.TR_SUCCESS is defined in the `mkl_rci.h` and `mkl_rci.fi` include files.

?trnlspbc_solve

Solves a nonlinear least squares problem with linear (bound) constraints using the Trust-Region algorithm.

Syntax

Fortran:

```
res = strnlspbc_solve(handle, fvec, fjac, RCI_Request)
res = dtrnlspbc_solve(handle, fvec, fjac, RCI_Request)
```

C:

```
MKL_INT strnlspbc_solve (_TRNSPBC_HANDLE_t* &handle, float* fvec, float* fjac, MKL_INT* &RCI_Request);
MKL_INT dtrnlspbc_solve (_TRNSPBC_HANDLE_t* &handle, double* fvec, double* fjac, MKL_INT* &RCI_Request);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The ?trnlspbc_solve routine, based on RCI, uses the Trust-Region algorithm to solve nonlinear least squares problems with linear (bound) constraints. The problem is stated as follows:

$$\min_{x \in \mathbb{R}^n} \|F(x)\|_2^2 = \min_{x \in \mathbb{R}^n} \|y - f(x)\|_2^2, y \in \mathbb{R}^m, x \in \mathbb{R}^n, f: \mathbb{R}^n \rightarrow \mathbb{R}^m, m \geq n$$

where

$$l_i \leq x_i \leq u_i$$

$$i = 1, \dots, n.$$

The *RCI_Request* parameter provides additional information:

RCI_Request Value	Description
2	Request to calculate the Jacobian matrix and put the result into <i>fjac</i>

RCI_Request Value	Description
1	Request to recalculate the function at vector x and put the result into $fvec$
0	One successful iteration step on the current trust-region radius (that does not mean that the value of x has changed)
-1	The algorithm has exceeded the maximum number of iterations
-2	$\Delta < \text{eps}(1)$
-3	$\ F(x)\ _2 < \text{eps}(2)$
-4	The Jacobian matrix is singular. $\ \mathcal{J}(x)_{(1:m,j)}\ _2 < \text{eps}(3), j = 1, \dots, n$
-5	$\ s\ _2 < \text{eps}(4)$
-6	$\ F(x)\ _2 - \ F(x) - \mathcal{J}(x)s\ _2 < \text{eps}(5)$

Note:

- $\mathcal{J}(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

<i>handle</i>	Type <code>_TRNSPBC_HANDLE_t</code> in C/C++ and <code>INTEGER*8</code> in FORTRAN.
<i>fvec</i>	<code>REAL</code> for <code>strnlspbc_solve</code> <code>DOUBLE PRECISION</code> for <code>dtrnlspbc_solve</code> Array of size m . Contains the function values at x , where $fvec(i) = (y_i - f_i(x))$.
<i>fjac</i>	<code>REAL</code> for <code>strnlspbc_solve</code> <code>DOUBLE PRECISION</code> for <code>dtrnlspbc_solve</code> Array of size m by n . Contains the Jacobian matrix of the function.

Output Parameters

<i>fvec</i>	<code>REAL</code> for <code>strnlspbc_solve</code> <code>DOUBLE PRECISION</code> for <code>dtrnlspbc_solve</code> Array of size m . Updated function evaluated at x .
<i>RCI_Request</i>	<code>INTEGER</code> . Informs about the task stage. See the Description section for the parameter values and their meaning.
<i>res</i>	<code>INTEGER</code> . Informs about the task completion. $res = \text{TR_SUCCESS}$ means the routine completed the task normally. <code>TR_SUCCESS</code> is defined in the <code>mkl_rci.h</code> and <code>mkl_rci.fi</code> include files.

?trnlspbc_get

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

Syntax**Fortran:**

```
res = strnlspbc_get(handle, iter, st_cr, r1, r2)
res = dtrnlspbc_get(handle, iter, st_cr, r1, r2)
```

C:

```
MKL_INT strnlspbc_get (_TRNSPBC_HANDLE_t* &handle, MKL_INT* &iter, MKL_INT* &st_cr,
float* &r1, float* &r2);

MKL_INT dtrnlspbc_get (_TRNSPBC_HANDLE_t* &handle, MKL_INT* &iter, MKL_INT* &st_cr,
double* &r1, double* &r2);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The routine retrieves the current number of iterations, the stop criterion, the initial residual, and final residual.

The *st_cr* parameter contains the stop criterion:

st_cr Value	Description
1	The algorithm has exceeded the maximum number of iterations
2	$\Delta < \text{eps}(1)$
3	$\ F(x)\ _2 < \text{eps}(2)$
4	The Jacobian matrix is singular. $\ \mathcal{J}(x)_{(1:m,j)}\ _2 < \text{eps}(3), j = 1, \dots, n$
5	$\ s\ _2 < \text{eps}(4)$
6	$\ F(x)\ _2 - \ F(x) - \mathcal{J}(x)s\ _2 < \text{eps}(5)$

Note:

- $\mathcal{J}(x)$ is the Jacobian matrix.
- Δ is the trust-region area.
- $F(x)$ is the value of the functional.
- s is the trial step.

Input Parameters

handle Type `_TRNSPBC_HANDLE_t` in C/C++ and `INTEGER*8` in FORTRAN.

Output Parameters

<i>iter</i>	INTEGER. Contains the current number of iterations.
<i>st_cr</i>	INTEGER. Contains the stop criterion. See the Description section for the parameter values and their meanings.
<i>r1</i>	
	REAL for strnlspbc_get DOUBLE PRECISION for dtrnlspbc_get Contains the residual, $(y - f(x))$ given the initial x .
<i>r2</i>	REAL for strnlspbc_get DOUBLE PRECISION for dtrnlspbc_get Contains the final residual, that is, the value of the function $(y - f(x))$ of the final x resulting from the algorithm operation.
<i>res</i>	INTEGER. Informs about the task completion. <i>res</i> = TR_SUCCESS - the routine completed the task normally. TR_SUCCESS is defined in the <code>mkl_rci.h</code> and <code>mkl_rci.fi</code> include files.

?trnlspbc_delete

Releases allocated data.

Syntax

Fortran:

```
res = strnlspbc_delete(handle)
res = dtrnlspbc_delete(handle)
```

C:

```
MKL_INT strnlspbc_delete (_TRNSPBC_HANDLE_t* &handle);
MKL_INT dtrnlspbc_delete (_TRNSPBC_HANDLE_t* &handle);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`
- C: `mkl_rci.h`

Description

The ?trnlspbc_delete routine releases all memory allocated for the handle.

NOTE

This routine flags memory as not used, but to actually release all memory you must call the support function `mkl_free_buffers`.

Input Parameters

handle Type `_TRNSPBC_HANDLE_t` in C/C++ and `INTEGER*8` in FORTRAN.

Output Parameters

res INTEGER. Informs about the task completion.

`res = TR_SUCCESS` means the routine completed the task normally.

`TR_SUCCESS` is defined in the `mkl_rci.h` and `mkl_rci.fi` include files.

Jacobian Matrix Calculation Routines

This section describes routines that compute the Jacobian matrix using the central difference algorithm. Jacobian matrix calculation is required to solve a nonlinear least squares problem and systems of nonlinear equations (with or without linear bound constraints). Routines for calculation of the Jacobian matrix have the "Black-Box" interfaces, where you pass the objective function via parameters. Your objective function must have a fixed interface.

Jacobian Matrix Calculation Routines

Routine Name	Operation
?jacobi_init	Initializes the solver.
?jacobi_solve	Computes the Jacobian matrix of the function on the basis of RCI using the central difference algorithm.
?jacobi_delete	Removes data.
?jacobi	Computes the Jacobian matrix of the <code>fcn</code> function using the central difference algorithm.
?jacobix	Presents an alternative interface for the ?jacobi function enabling you to pass additional data into the objective function.

?jacobi_init

Initializes the solver for Jacobian calculations.

Syntax

Fortran:

```
res = sjacobi_init(handle, n, m, x, fjac, eps)
res = djacobi_init(handle, n, m, x, fjac, eps)
```

C:

```
MKL_INT sjacobi_init (_JACOBIMATRIX_HANDLE_t* &handle, MKL_INT* &n, MKL_INT* &m, float* x, float* fjac, float* &eps);
MKL_INT djacobi_init (_JACOBIMATRIX_HANDLE_t* &handle, MKL_INT* &n, MKL_INT* &m, double* x, double* fjac, double* &eps);
```

Include Files

- Fortran: `mkl_rci.fi`
- Fortran 90: `mkl_rci.f90`

- C: mkl_rci.h

Description

The routine initializes the solver.

Input Parameters

<i>n</i>	INTEGER. Length of <i>x</i> .
<i>m</i>	INTEGER. Length of <i>F</i> .
<i>x</i>	REAL for sjacobi_init DOUBLE PRECISION for djacobi_init Array of size <i>n</i> . Vector, at which the function is evaluated.
<i>eps</i>	REAL for sjacobi_init DOUBLE PRECISION for djacobi_init Precision of the Jacobian matrix calculation.
<i>fjac</i>	REAL for sjacobi_init DOUBLE PRECISION for djacobi_init Array of size (<i>m,n</i>). Contains the Jacobian matrix of the function.

Output Parameters

<i>handle</i>	Data object of the _JACOBIMATRIX_HANDLE_t type in C/C++ and INTEGER*8 in FORTRAN.
<i>res</i>	INTEGER. Indicates task completion status. <ul style="list-style-type: none"> • <i>res</i> = TR_SUCCESS - the routine completed the task normally. • <i>res</i> = TR_INVALID_OPTION - there was an error in the input parameters. • <i>res</i> = TR_OUT_OF_MEMORY - there was a memory error. TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.fi (Fortran) and mkl_rci.h (C) include files.

?jacobi_solve

Computes the Jacobian matrix of the function using RCI and the central difference algorithm.

Syntax

Fortran:

```
res = sjacobi_solve(handle, f1, f2, RCI_Request)
res = djacobi_solve(handle, f1, f2, RCI_Request)
```

C:

```
MKL_INT sjacobi_solve (_JACOBIMATRIX_HANDLE_t* &handle, float* f1, float* f2, MKL_INT*
&RCI_Request);
```

```
MKL_INT djacobi_solve (_JACOBIMATRIX_HANDLE_t* &handle, double* f1, double* f2,
MKL_INT* &RCI_Request);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The ?jacobi_solve routine computes the Jacobian matrix of the function using RCI and the central difference algorithm.

See usage examples in Fortran and C in the examples\solverf\source and examples\solverc\source folders of your Intel MKL directory, respectively. Specifically, see sjacobi_rci_f.f and djacobi_rci_f.f for Fortran and sjacobi_rci_c.c and djacobi_rci_c.c for C.

Input Parameters

handle Type _JACOBIMATRIX_HANDLE_t in C/C++ and INTEGER*8 in FORTRAN.

Output Parameters

<i>f1</i>	REAL for sjacobi_solve DOUBLE PRECISION for djacobi_solve Contains the updated function values at $x + \text{eps}$.
<i>f2</i>	REAL for sjacobi_solve DOUBLE PRECISION for djacobi_solve Array of size m . Contains the updated function values at $x - \text{eps}$.
<i>RCI_Request</i>	INTEGER. Informs about the task completion. When equal to 0, the task has completed successfully. $RCI_Request = 1$ indicates that you should compute the function values at the current x point and put the results into <i>f1</i> . $RCI_Request = 2$ indicates that you should compute the function values at the current x point and put the results into <i>f2</i> .
<i>res</i>	INTEGER. Indicates the task completion status. <ul style="list-style-type: none"> • <i>res</i> = TR_SUCCESS - the routine completed the task normally. • <i>res</i> = TR_INVALID_OPTION - there was an error in the input parameters. TR_SUCCESS and TR_INVALID_OPTION are defined in the mkl_rci.fi (Fortran) and mkl_rci.h (C) include files.

See Also

?jacobi_init

?jacobi_delete

Releases allocated data.

Syntax

Fortran:

```
res = sjacobi_delete(handle)
res = djacobi_delete(handle)
```

C:

```
MKL_INT sjacobi_delete (_JACOBIMATRIX_HANDLE_t* &handle);
MKL_INT djacobi_delete (_JACOBIMATRIX_HANDLE_t* &handle);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The ?jacobi_delete routine releases all memory allocated for the handle.

This routine flags memory as not used, but to actually release all memory you must call the support function [mkl_free_buffers](#).

Input Parameters

handle Type _JACOBIMATRIX_HANDLE_t in C/C++ and INTEGER*8 in FORTRAN.

Output Parameters

res INTEGER. Informs about the task completion.

res = TR_SUCCESS means the routine completed the task normally.

TR_SUCCESS is defined in the mkl_rci.h and mkl_rci.fi include files.

?jacobi

Computes the Jacobian matrix of the objective function using the central difference algorithm.

Syntax

Fortran:

```
res = sjacobi(fcn, n, m, fjac, x, eps)
res = djacobi(fcn, n, m, fjac, x, eps)
```

C:

```
MKL_INT sjacobi (USRFCNS fcn, MKL_INT* &n, MKL_INT* &m, float* fjac, float* x, float* &eps);
MKL_INT djacobi (USRFCND fcn, MKL_INT* &n, MKL_INT* &m, double* fjac, double* x, double* &eps);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The ?jacobi routine computes the Jacobian matrix for function *fcn* using the central difference algorithm. This routine has a "Black-Box" interface, where you input the objective function via parameters. Your objective function must have a fixed interface.

See calling and usage examples in FORTRAN and C in the examples\solverf\source and examples\solverc\source folders of your Intel MKL directory, respectively. Specifically, see ex_nlsqp_f.f and ex_nlsqp_bc_f.f for Fortran and ex_nlsqp_c.c and ex_nlsqp_bc_c.c for C.

Input Parameters

fcn User-supplied subroutine to evaluate the function that defines the least squares problem. Call *fcn* (*m*, *n*, *x*, *f*) with the following parameters:

Parameter	Type	Description
Input Parameters		
<i>m</i>	INTEGER	Length of <i>f</i>
<i>n</i>	INTEGER	Length of <i>x</i>
<i>x</i>	REAL for sjacobi DOUBLE PRECISION for djacobi	Array of size <i>n</i> . Vector, at which the function is evaluated. The <i>fcn</i> function should not change this parameter.
Output Parameters		
<i>f</i>	REAL for sjacobix DOUBLE PRECISION for djacobix	Array of size <i>m</i> ; contains the function values at <i>x</i> .

You need to declare *fcn* as EXTERNAL in the calling program.

<i>n</i>	INTEGER. Length of <i>X</i> .
<i>m</i>	INTEGER. Length of <i>F</i> .
<i>x</i>	REAL for sjacobi DOUBLE PRECISION for djacobi Array of size <i>n</i> . Vector at which the function is evaluated.
<i>eps</i>	REAL for sjacobi DOUBLE PRECISION for djacobi Precision of the Jacobian matrix calculation.

Output Parameters

fjac REAL for sjacobi
 DOUBLE PRECISION for djacobi
 Array of size (*m,n*). Contains the Jacobian matrix of the function.

res INTEGER. Indicates task completion status.

- *res* = TR_SUCCESS - the routine completed the task normally.
- *res* = TR_INVALID_OPTION - there was an error in the input parameters.
- *res* = TR_OUT_OF_MEMORY - there was a memory error.

TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.fi (Fortran) and mkl_rci.h (C) include files.

See Also

?jacobix

?jacobix

Alternative interface for ?jacobi function for passing additional data into the objective function.

Syntax

Fortran:

```
res = sjacobix(fcn, n, m, fjac, x, eps, user_data)
res = djacobix(fcn, n, m, fjac, x, eps, user_data)
```

C:

```
MKL_INT sjacobix (USRFCNXS fcn, MKL_INT* &n, MKL_INT * &m, float* fjac, float* x,
float* &eps, void* user_data);

MKL_INT djacobix (USRFCNXD fcn, MKL_INT* &n, MKL_INT * &m, double* fjac, double* x,
double* &eps, void* user_data);
```

Include Files

- Fortran: mkl_rci.fi
- Fortran 90: mkl_rci.f90
- C: mkl_rci.h

Description

The ?jacobix routine presents an alternative interface for the ?jacobi function that enables you to pass additional data into the objective function *fcn*.

See calling and usage examples in Fortran and C in the examples\solverf\source and examples\solverc\source folders of your Intel MKL directory, respectively. Specifically, see ex_nlsqp_f90_x.f90 and ex_nlsqp_bc_f90_x.f90 for Fortran and ex_nlsqp_c_x.c and ex_nlsqp_bc_c_x.c for C.

Input Parameters

fcn User-supplied subroutine to evaluate the function that defines the least squares problem. Call *fcn* (*m, n, x, f, user_data*) with the following parameters:

Parameter	Type	Description
Input Parameters		
<i>m</i>	INTEGER	Length of <i>f</i>
<i>n</i>	INTEGER	Length of <i>x</i>
<i>x</i>	REAL for sjacobix DOUBLE PRECISION for djacobix	Array of size <i>n</i> . Vector, at which the function is evaluated. The <i>fcn</i> function should not change this parameter.
<i>user_data</i>	INTEGER(C_INTPTR_T) , for Fortran void*, for C	(Fortran) Reference to your additional data, if any, passed by value: <i>user_data</i> =%VAL(LOC(<i>data</i>)). Otherwise, a dummy argument. (C) Pointer to your additional data, if any. Otherwise, a dummy argument.
Output Parameters		
<i>f</i>	REAL for sjacobix DOUBLE PRECISION for djacobix	Array of size <i>m</i> ; contains the function values at <i>x</i> .

You need to declare *fcn* as EXTERNAL in the calling program.

<i>n</i>	INTEGER. Length of <i>X</i> .
<i>m</i>	INTEGER. Length of <i>F</i> .
<i>x</i>	REAL for sjacobix DOUBLE PRECISION for djacobix Array of size <i>n</i> . Vector at which the function is evaluated.
<i>eps</i>	REAL for sjacobix DOUBLE PRECISION for djacobix Precision of the Jacobian matrix calculation.
<i>user_data</i>	(Fortran) INTEGER(C_INTPTR_T). Reference to your additional data, passed by value: <i>user_data</i> =%VAL(LOC(<i>data</i>)). Otherwise, a dummy argument. (C) void*. Pointer to your additional data. If there is no additional data, this is a dummy argument.

Output Parameters

<i>fjac</i>	REAL for sjacobix DOUBLE PRECISION for djacobix Array of size (<i>m,n</i>). Contains the Jacobian matrix of the function.
<i>res</i>	INTEGER. Indicates task completion status.

- `res = TR_SUCCESS` - the routine completed the task normally.
- `res = TR_INVALID_OPTION` - there was an error in the input parameters.
- `res = TR_OUT_OF_MEMORY` - there was a memory error.

`TR_SUCCESS`, `TR_INVALID_OPTION`, and `TR_OUT_OF_MEMORY` are defined in the `mkl_rci.fi` (Fortran) and `mkl_rci.h` (C) include files.

See Also

[?jacobi](#)

Support Functions

Intel® Math Kernel Library (Intel® MKL) support functions are subdivided into the following groups according to their purpose:

[Version Information](#)

[Threading Control](#)

[Error Handling](#)

[Character Equality Testing](#)

[Timing](#)

[Memory Management](#)

[Single Dynamic Library Control](#)

[Intel® Many Integrated Core \(Intel® MIC\) Architecture Support](#)

[Conditional Numerical Reproducibility Control](#)

[Miscellaneous](#)

The following table lists Intel MKL support functions.

Intel MKL Support Functions

Function Name	Operation
Version Information	
<code>mkl_get_version</code>	A C function that returns the Intel MKL version.
<code>mkl_get_version_string</code>	Returns the Intel MKL version in a character string.
Threading Control	
<code>mkl_set_num_threads</code>	Specifies the number of threads to use.
<code>mkl_domain_set_num_threads</code>	Specifies the number of threads for a particular function domain.
<code>mkl_set_num_threads_local</code>	Specifies the number of threads for all Intel MKL functions on the current execution thread.
<code>mkl_set_dynamic</code>	Enables Intel MKL to dynamically change the number of threads.
<code>mkl_get_max_threads</code>	Gets the number of threads targeted for parallelism.
<code>mkl_domain_get_max_threads</code>	Gets the number of threads targeted for parallelism for a particular function domain.
<code>mkl_get_dynamic</code>	Determines whether Intel MKL is enabled to dynamically change the number of threads.
Error Handling	
<code>xerbla</code>	Handles error conditions for the BLAS, LAPACK, VSL, VML routines.
<code>pixerbla</code>	Handles error conditions for the ScaLAPACK routines.

Function Name	Operation
<code>mkl_set_exit_handler</code>	Sets the custom handler of fatal errors.
Character Equality Testing	
<code>lsame</code>	Tests two characters for equality regardless of the case.
<code>lsamen</code>	Tests two character strings for equality regardless of the case.
Timing	
<code>second/dsecnd</code>	Returns elapsed time in seconds. Use to estimate real time between two calls to this function.
<code>mkl_get_cpu_clocks</code>	Returns elapsed CPU clocks.
<code>mkl_get_cpu_frequency</code>	Returns CPU frequency value in GHz.
<code>mkl_get_max_cpu_frequency</code>	Returns the maximum CPU frequency value in GHz.
<code>mkl_get_clocks_frequency</code>	Returns the frequency value in GHz based on constant-rate Time Stamp Counter.
Memory Management	
<code>mkl_free_buffers</code>	Frees unused memory allocated by the Intel MKL Memory Allocator.
<code>mkl_thread_free_buffers</code>	Frees unused memory allocated by the Intel MKL Memory Allocator in the current thread.
<code>mkl_mem_stat</code>	Reports the status of the Intel MKL Memory Allocator.
<code>mkl_peak_mem_usage</code>	Reports the peak memory allocated by the Intel MKL Memory Allocator.
<code>mkl_disable_fast_mm</code>	Turns off the Intel MKL Memory Allocator for Intel MKL functions to directly use the system <code>malloc/free</code> functions.
<code>mkl_malloc</code>	Allocates an aligned memory buffer.
<code>mkl_calloc</code>	Allocates and initializes an aligned memory buffer.
<code>mkl_realloc</code>	Changes the size of memory buffer allocated by <code>mkl_malloc/mkl_calloc</code> .
<code>mkl_free</code>	Frees the aligned memory buffer allocated by <code>mkl_malloc/mkl_calloc</code> .
Single Dynamic Library (SDL) Control	
<code>mkl_set_interface_layer</code>	Sets the interface layer for Intel MKL at run time.
<code>mkl_set_threading_layer</code>	Sets the threading layer for Intel MKL at run time.
<code>mkl_set_xerbla</code>	Replaces the error handling routine. Use with SDL on Windows* OS.
<code>mkl_set_progress</code>	Replaces the progress information routine. Use with SDL on Windows* OS.
Intel MIC Architecture Support	

Function Name	Operation
<code>mkl_mic_enable</code>	Enables Automatic Offload mode.
<code>mkl_mic_disable</code>	Disables Automatic Offload mode.
<code>mkl_mic_get_device_count</code>	Returns the number of Intel® Xeon Phi™ coprocessors on the system when called on the host CPU.
<code>mkl_mic_set_workdivision</code>	For computations in the Automatic Offload mode, sets the fraction of the work for the specified coprocessor or host CPU to do.
<code>mkl_mic_get_workdivision</code>	For computations in the Automatic Offload mode, retrieves the fraction of the work for the specified coprocessor or host CPU to do.
<code>mkl_mic_set_max_memory</code>	Sets the maximum amount of coprocessor memory reserved for Automatic Offload computations.
<code>mkl_mic_free_memory</code>	Frees the coprocessor memory reserved for Automatic Offload computations.
<code>mkl_mic_register_memory</code>	Enables/disables the <code>mkl_malloc</code> function running in Automatic Offload mode to register allocated memory.
<code>mkl_mic_set_device_num_threads</code>	Sets the maximum number of threads to use on an Intel Xeon Phi coprocessor for the Automatic Offload computations.
<code>mkl_mic_set_resource_limit</code>	For computations in the Automatic Offload mode, sets the maximum fraction of available Intel Xeon Phi coprocessor computational resources (cores) that the calling process can use.
<code>mkl_mic_get_resource_limit</code>	For computations in the Automatic Offload mode, retrieves the maximum fraction of available Intel Xeon Phi coprocessor computational resources (cores) that the calling process can use.
<code>mkl_mic_set_offload_report</code>	Turns on/off reporting of Automatic Offload profiling.
Conditional Numerical Reproducibility (CNR) Control	
<code>mkl_cbwr_set</code>	Configures the CNR mode of Intel MKL.
<code>mkl_cbwr_get</code>	Returns the current CNR settings.
<code>mkl_cbwr_get_auto_branch</code>	Automatically detects the CNR code branch for your platform.
Miscellaneous	
<code>mkl_progress</code>	Tracks computational progress of selective Intel MKL routines.
<code>mkl_enable_instructions</code>	Enables Intel MKL to dispatch a new Intel® architecture on hardware (or simulation) supporting the appropriate instruction set.
<code>mkl_set_env_mode</code>	Sets up the mode that ignores environment settings specific to Intel MKL.
<code>mkl_verbose</code>	Enables or disables Intel MKL Verbose mode.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Using a Fortran Interface Module for Support Functions

To call a support function from your Fortran application, include one of the following statements in your code:

- INCLUDE mkl_service.fi or INCLUDE mkl.fi
- USE mkl_service

The USE statement references the mkl_service.mod interface module corresponding to your architecture and programming interface. The module provides an application programming interface to Intel MKL support entities, such as subroutines and constants. Because Fortran interface modules are compiler-dependent, Intel MKL offers the mkl_service.f90 source file for the module, as well as architecture-specific and interface-specific mkl_service modules precompiled with the Intel® Fortran or Intel® Visual Fortran compiler. These modules are available in the following subdirectories of the Intel MKL include directory:

Architecture, Interface	Subdirectory of the Intel MKL Installation Directory
IA-32	include\ia32
Intel® 64, LP64	include\intel64\lp64
Intel® 64, ILP64	include\intel64\ilp64

To ensure that your application searches the right module, specify the appropriate subdirectory during compilation as an additional directory for the include path (through the /I option on Windows* OS or the -I option on Linux* OS or OS X*).

If you are using a non-Intel Fortran compiler, you need to build the module yourself by compiling the mkl_service.f90 file, available in the Intel MKL include directory.

For more information on compiler-dependent functions and modules, refer to the *Intel MKL User's Guide*.

Version Information

Intel® MKL provides methods for extracting information about the library version number, such as:

- extracting a version string using the mkl_get_version_string function
- using the mkl_get_version function to obtain an MKLVersion structure that contains the version information

A makefile is also provided to automatically build the examples and output summary files containing the version information for the current library.

mkl_get_version

A C function that returns the Intel MKL version.

Syntax

```
void mkl_get_version( MKLVersion* pVersion );
```

Include Files

- mkl.h

Output Parameters

pVersion Pointer to the MKLVersion structure.

Description

The `mkl_get_version` function collects information about the active C version of the Intel MKL software and returns this information in a structure of `MKLVersion` type by the *pVersion* address. The `MKLVersion` structure type is defined in the `mkl_types.h` file. The following fields of the `MKLVersion` structure are available:

MajorVersion	is the major number of the current library version.
MinorVersion	is the minor number of the current library version.
UpdateVersion	is the update number of the current library version.
ProductStatus	is the status of the current library version. Possible variants are "Beta" or "Product".
Build	is the string that contains the build date and the internal build number.
Platform	is the string that contains the current architecture. Possible variants are "IA-32 architecture" or "Intel(R) 64 architecture".
Processor	is the processor optimization. Normally it is targeted for the processor installed on your system and based on the detection of the Intel MKL library that is optimal for the installed processor. In the Conditional Numerical Reproducibility (CNR) mode, the processor optimization matches the selected CNR branch.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

`mkl_get_version` Usage

```
#include <stdio.h>
#include <stdlib.h>
#include "mkl_service.h"
```

```

int main(void)
{
    MKLVersion Version;

    mkl_get_version(&Version);

    printf("Major version:          %d\n", Version.MajorVersion);
    printf("Minor version:         %d\n", Version.MinorVersion);
    printf("Update version:        %d\n", Version.UpdateVersion);
    printf("Product status:       %s\n", Version.ProductStatus);
    printf("Build:                 %s\n", Version.Build);
    printf("Platform:              %s\n", Version.Platform);
    printf("Processor optimization: %s\n", Version.Processor);
    printf("=====================\n");
    printf("\n");

    return 0;
}

```

Output:

Major Version	11
Minor Version	0
Update Version	2
Product status	Product
Build	20121113
Platform	Intel(R) 64 architecture
Processor optimization	Intel(R) Core(TM) i7 Processor

See Also

[Conditional Numerical Reproducibility Control](#)

[mkl_get_version_string](#)

Returns the Intel MKL version in a character string.

Syntax

Fortran:

```
call mkl_get_version_string( buf )
```

C:

```
void mkl_get_version_string (char* buf, int len);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Output Parameters

Name	Type	Description
<i>buf</i>	FORTRAN: CHARACTER*198 C: char*	Source string
C: <i>len</i>	int	Length of the source string

Description

The function returns a string that contains the Intel MKL version.

For usage details, see the code example below:

Example

Fortran

```
program mkl_get_version_string
character*198  buf
call mkl_get_version_string(buf)
write(*,'(a)') buf
end
```

C

```
#include <stdio.h>
#include "mkl_service.h"

int main(void)
{
    int len=198;
    char buf[198];
    mkl_get_version_string(buf, len);
    printf("%s\n",buf);
    printf("\n");
    return 0;
}
```

Threading Control

Intel® MKL computational functions can create internal regions for parallel execution. Because Intel MKL operates within an OpenMP* execution environment, you can control the number of threads for Intel MKL using OpenMP* run-time library routines and environment variables (see the OpenMP* specification for details). Additionally Intel MKL provides *optional* threading control functions and environment variables that enable you to specify the number of threads for Intel MKL and to control dynamic adjustment of the number of threads *independently* of the OpenMP* settings. The settings made with the Intel MKL threading control functions and environment variables do not affect OpenMP* settings but take precedence over them.

This section describes how to control the number of threads using Intel MKL functions for threading control. If none of the threading control functions is used, Intel MKL environment variables may control Intel MKL threading. For details of those environment variables, see the *Intel MKL User's Guide*.

You can specify the number of threads for Intel MKL function domains with the `mkl_set_num_threads` or `mkl_domain_set_num_threads` function. While `mkl_set_num_threads` specifies the number of threads for the entire Intel MKL, `mkl_domain_set_num_threads` does it for a specific function domain. The following table lists the function domains that support independent threading control. The table also provides named constants to pass to threading control functions as a parameter that specifies the function domain.

Intel MKL Function Domains

Function Domain	Named Constant
Basic Linear Algebra Subroutines (BLAS)	<code>MKL_DOMAIN_BLAS</code>
Fast Fourier Transform (FFT) functions, except Cluster FFT functions	<code>MKL_DOMAIN_FFT</code>
Vector Math Library (VML) Functions	<code>MKL_DOMAIN_VML</code>
Parallel Direct Solver (PARDISO) functions	<code>MKL_DOMAIN_PARDISO</code>
All Intel MKL functions except the functions from the domains where the number of threads is set explicitly	<code>MKL_DOMAIN_ALL</code>

WARNING

Do not increase the number of threads used for Intel MKL PARDISO between the first call to `pardiso` and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of threads, increasing it after the initial call can cause incorrect results.

Both `mkl_set_num_threads` and `mkl_domain_set_num_threads` functions set the number of threads globally, which means that the setting applies to all execution threads. Use the `mkl_set_num_threads_local` function to specify different numbers of threads for Intel MKL on different execution threads of your application. The thread-local settings take precedence over the global settings. However, the thread-local settings may have undesirable side effects (see the description of the `mkl_set_num_threads_local` function for details).

By default, Intel MKL can adjust the specified number of threads dynamically. For example, Intel MKL may use fewer threads if the size of the computation is not big enough or not create parallel regions when running within an OpenMP* parallel region. Although Intel MKL may actually use a different number of threads from the number specified, the library does not create parallel regions with more threads than specified. If dynamic adjustment of the number of threads is disabled, Intel MKL attempts to use the specified number of threads in internal parallel regions (for more information, see the *Intel MKL User's Guide*). Use the `mkl_set_dynamic` function to control dynamic adjustment of the number of threads.

`mkl_set_num_threads`

Specifies the number of threads to use.

Syntax

Fortran:

```
call mkl_set_num_threads( nt )
```

C:

```
void mkl_set_num_threads( int nt );
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>nt</i>	FORTRAN: INTEGER C: int	$nt > 0$ - The number of threads suggested by the user. $nt \leq 0$ - Invalid value, which is ignored.

Description

This function allows you to specify how many threads Intel MKL should use for internal parallel regions. If this number is not set (default), Intel MKL functions use the OpenMP* number of threads. The specified number of threads applies:

- To all Intel MKL functions except the functions from the domains where the number of threads is set with [mkl_domain_set_num_threads](#)
- To all execution threads except the threads where the number of threads is set with [mkl_set_num_threads_local](#)

The number specified is a hint, and Intel MKL may actually use a different number.

NOTE

This function takes precedence over the `MKL_NUM_THREADS` environment variable.

Example

```
// C/C++
#include "mkl_service.h"
...
mkl_set_num_threads(4);
my_compute_using_mkl(); // Intel MKL uses up to 4 threads

! Fortran
use mkl_service
...
call mkl_set_num_threads(4)
call my_compute_using_mkl !Intel MKL uses up to 4 threads
```

mkl_domain_set_num_threads

Specifies the number of threads for a particular function domain.

Syntax

Fortran:

```
ierr = mkl_domain_set_num_threads( nt, domain )
```

C:

```
int mkl_domain_set_num_threads (int nt, int domain);
```

Fortran Include Files/Modules

- Include file: mkl.fi

- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>nt</i>	FORTRAN: INTEGER C: int	<i>nt</i> > 0 - The number of threads suggested by the user. <i>nt</i> = 0 - A request to reset the number of threads and use the OpenMP* number of threads. <i>nt</i> < 0 - Invalid value, which is ignored.
<i>domain</i>	FORTRAN: INTEGER C: int	The named constant that defines the targeted domain.

Description

This function specifies how many threads a particular function domain of Intel MKL should use. If this number is not set (default) or if it is set to zero in a call to this function, Intel MKL will use the OpenMP* number of threads. The number of threads specified applies to the specified function domain on all execution threads except the threads where the number of threads is set with [mkl_set_num_threads_local](#). For a list of supported values of the *domain* argument, see [Table "Intel MKL Function Domains"](#).

The number of threads specified is only a hint, and Intel MKL may actually use a different number.

NOTE

This function takes precedence over the `MKL_DOMAIN_NUM_THREADS` environment variable.

Return Values

Name	Type	Description
<i>ierr</i>	FORTRAN: INTEGER C: int	1 - Indicates no error, execution is successful. 0 - Indicates a failure, possibly because of invalid input parameters.

Example

```
// C/C++
#include "mkl_service.h"
...
mkl_domain_set_num_threads(4, MKL_DOMAIN_BLAS);
my_compute_using_mkl blas(); // Intel MKL BLAS functions use up to 4 threads
my_compute_using_mkl dft(); // Intel MKL FFT functions use the default number of threads
```

```
! Fortran
use mkl_service
integer(4) :: status
...
status = mkl_domain_set_num_threads(4, MKL_DOMAIN_BLAS)
call my_compute_with_mkl blas() !Intel MKL BLAS functions use up to 4 threads
call my_compute_with_mkl dft() !Intel MKL FFT functions use the default number of threads
```

mkl_set_num_threads_local

Specifies the number of threads for all Intel MKL functions on the current execution thread.

Syntax

Fortran:

```
save_nt = mkl_set_num_threads_local( nt )
```

C:

```
int mkl_set_num_threads_local( int nt );
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>nt</i>	FORTRAN: INTEGER*4 C: int	<p><i>nt</i> > 0 - The number of threads for Intel MKL functions to use on the current execution thread.</p> <p><i>nt</i> = 0 - A request to reset the thread-local number of threads and use the global number.</p>

Description

This function sets the number of threads that Intel MKL functions should request for parallel computation. The number of threads is thread-local, which means that it only affects the current execution thread of the application. If the thread-local number is not set or if this number is set to zero in a call to this function, Intel MKL functions use the global number of threads. You can set the global number of threads using the [mkl_set_num_threads](#) or [mkl_domain_set_num_threads](#) function.

The thread-local number of threads takes precedence over the global number: if the thread-local number is non-zero, changes to the global number of threads have no effect on the current thread.

CAUTION

If your application is threaded with OpenMP* and parallelization of Intel MKL is based on nested OpenMP parallelism, different OpenMP parallel regions reuse OpenMP threads. Therefore a thread-local setting in one OpenMP parallel region may continue to affect not only the master thread after the parallel region ends, but also subsequent parallel regions. To avoid performance implications of this side effect, reset the thread-local number of threads before leaving the OpenMP parallel region (see [Examples](#) for how to do it).

Return Values

Name	Type	Description
save_nt	FORTRAN: INTEGER*4 C: int	The value of the thread-local number of threads that was used before this function call. Zero means that the global number of threads was used.

Examples

This example shows how to avoid the side effect of a thread-local number of threads by reverting to the global setting:

```
// C/C++
#include "omp.h"
#include "mkl_service.h"
...
mkl_set_num_threads(16);
my_compute_using_mkl();           // Intel MKL functions use up to 16 threads
#pragma omp parallel num_threads(2)
{
    if (0 == omp_get_thread_num())
        mkl_set_num_threads_local(4);
    else
        mkl_set_num_threads_local(12);

    my_compute_using_mkl();       // Intel MKL functions use up to 4 threads on thread 0
//    and up to 12 threads on thread 1
}
my_compute_using_mkl();           // Intel MKL functions use up to 4 threads (!)
mkl_set_num_threads_local( 0 );   // make master thread use global setting
my_compute_using_mkl();           // Intel MKL functions use up to 16 threads
```

```
! Fortran
use omp_lib
use mkl_service
integer(4) :: dummy
...
call mkl_set_num_threads(16)
call my_compute_using_mkl()      ! Intel MKL functions use up to 16 threads
 !$omp parallel num_threads(2)
    if (0 == omp_get_thread(num))  dummy = mkl_set_num_threads_local(4)
    if (1 == omp_get_thread(num))  dummy = mkl_set_num_threads_local(12)
    call my_compute_using_mkl()    ! Intel MKL functions use up to 4 threads on thread 0
                                  ! and up to 12 threads on thread 1
 !$omp end parallel
call my_compute_using_mkl()      ! Intel MKL functions use up to 4 threads (!)
dummy = mkl_set_num_threads_local(0) ! make master thread use global setting
call my_compute_using_mkl()      ! Now Intel MKL functions use up to 16 threads
```

This example shows how to avoid the side effect of a thread-local number of threads by saving and restoring the existing setting:

```
// C/C++
#include "mkl_service.h"
void my_compute( int nt )
{
    int save = mkl_set_num_threads_local( nt );    // save the Intel MKL number of threads
    my_compute_using_mkl();                         // Intel MKL functions use up to nt threads on this thread
    mkl_set_num_threads_local( save );              // restore the Intel MKL number of threads
}
```

```

! Fortran
subroutine my_compute(nt)
  use mkl_service
  integer(4) :: nt, save
  save = mkl_set_num_threads_local( nt )      ! save the Intel MKL number of threads
  call my_compute_using_mkl()      ! Intel MKL functions use up to nt threads on this thread
  save = mkl_set_num_threads_local( save )      ! restore the Intel MKL number of threads
end subroutine my_compute

```

mkl_set_dynamic

Enables Intel MKL to dynamically change the number of threads.

Syntax

Fortran:

```
call mkl_set_dynamic( flag )
```

C:

```
void mkl_set_dynamic (int flag);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
flag	<small>FORTRAN:</small> INTEGER <small>C:</small> int	<small>flag</small> = 0 - Requests disabling dynamic adjustment of the number of threads. <small>flag</small> ≠ 0 - Requests enabling dynamic adjustment of the number of threads.

Description

This function indicates whether Intel MKL can dynamically change the number of threads or should avoid doing this. The setting applies to all Intel MKL functions on all execution threads. This function takes precedence over the `MKL_DYNAMIC` environment variable.

Dynamic adjustment of the number of threads is enabled by default. Specifically, Intel MKL may use fewer threads in parallel regions than the number returned by the [mkl_get_max_threads](#) function. Disabling dynamic adjustment of the number of threads does not ensure that Intel MKL actually uses the specified number of threads, although the library attempts to use that number.

TIP

If you call Intel MKL from within an OpenMP* parallel region and want to create internal parallel regions, either disable dynamic adjustment of the number of threads or set the thread-local number of threads (see [mkl_set_num_threads_local](#) for how to do it).

Example

```
// C/C++
#include "mkl_service.h"
...
mkl_set_num_threads( 8 );
#pragma omp parallel
{
    my_compute_with_mkl();      // Intel MKL uses 1 thread, being called from OpenMP parallel region
    mkl_set_dynamic( 0 );      // disable adjustment of the number of threads
    my_compute_with_mkl();      // Intel MKL uses 8 threads
}

! Fortran
use mkl_service
...
call mkl_set_num_threads( 8 )
 !$omp parallel
    call my_compute_with_mkl      ! Intel MKL uses 1 thread, being called from OpenMP parallel region
    call mkl_set_dynamic(0)       ! disable adjustment of the number of threads
    call my_compute_with_mkl     ! Intel MKL uses 8 threads
 !$omp end parallel
```

mkl_get_max_threads

Gets the number of threads targeted for parallelism.

Syntax

Fortran:

```
nt = mkl_get_max_threads()
```

C:

```
int mkl_get_max_threads (void);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Description

This function returns the number of threads for Intel MKL to use in internal parallel regions. This number depends on whether dynamic adjustment of the number of threads by Intel MKL is disabled (by an environment setting or in a function call):

- If the dynamic adjustment is disabled, the function inspects the environment settings and return values of the function calls below in the order they are listed until it finds a non-zero value:
 - A call to `mkl_set_num_threads_local`
 - The last of the calls to `mkl_set_num_threads` or `mkl_domain_set_num_threads(..., MKL_DOMAIN_ALL)`
 - The `MKL_DOMAIN_NUM_THREADS` environment variable with the `MKL_DOMAIN_ALL` tag
 - The `MKL_NUM_THREADS` environment variable
 - A call to `omp_set_num_threads`

- The `OMP_NUM_THREADS` environment variable
- If the dynamic adjustment is enabled, the function returns the number of physical cores on your system.

The number of threads returned by this function is a hint, and Intel MKL may actually use a different number.

Return Values

Name	Type	Description
<code>nt</code>	<code>FORTRAN: INTEGER*4</code> <code>C: int</code>	The maximum number of threads for Intel MKL functions to use in internal parallel regions.

Example

```
// C/C++
#include "mkl_service.h"
...
if (1 == mkl_get_max_threads()) puts("Intel MKL does not employ threading");

! Fortran
use mkl_service
...
if (1 == mkl_get_max_threads()) print *, "Intel MKL does not employ threading"
```

See Also

[mkl_set_dynamic](#)
[mkl_get_dynamic](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_domain_get_max_threads

Gets the number of threads targeted for parallelism for a particular function domain.

Syntax

Fortran:

```
nt = mkl_domain_get_max_threads( domain )
```

C:

```
int mkl_domain_get_max_threads (int domain);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>domain</code>	<code>FORTRAN: INTEGER</code>	The named constant that defines the targeted domain.

Name	Type	Description
	C: int	

Description

Computational functions of the Intel MKL function domain defined by the *domain* parameter use the value returned by this function as a limit of the number of threads they should request for parallel computations. The `mkl_domain_get_max_threads` function returns the thread-local number of threads or, if that value is zero or not set, the global number of threads. To determine this number, the function inspects the environment settings and return values of the function calls below in the order they are listed until it finds a non-zero value:

- A call to `mkl_set_num_threads_local`
- The last of the calls to `mkl_set_num_threads` or `mkl_domain_set_num_threads(..., MKL_DOMAIN_ALL)`
- A call to `mkl_domain_set_num_threads(..., domain)`
- The `MKL_DOMAIN_NUM_THREADS` environment variable with the `MKL_DOMAIN_ALL` tag
- The `MKL_DOMAIN_NUM_THREADS` environment variable (with the specific domain tag)
- The `MKL_NUM_THREADS` environment variable
- A call to `omp_set_num_threads`
- The `OMP_NUM_THREADS` environment variable

Actual number of threads used by the Intel MKL computational functions may vary depending on the problem size and on whether dynamic adjustment of the number of threads is enabled (see the description of `mkl_set_dynamic`). For a list of supported values of the *domain* argument, see [Table "Intel MKL Function Domains"](#).

Return Values

Name	Type	Description
<i>nt</i>	FORTRAN: INTEGER*4 C: int	The maximum number of threads for Intel MKL functions from a given domain to use in internal parallel regions. If an invalid value of <i>domain</i> is supplied, the function returns the number of threads for <code>MKL_DOMAIN_ALL</code>

Example

```
// C/C++
#include "mkl_service.h"
...
if (1 < mkl_domain_get_max_threads(MKL_DOMAIN_BLAS))
puts("Intel MKL BLAS functions employ threading");

! Fortran
use mkl_service
...
if (1 < mkl_domain_get_max_threads(MKL_DOMAIN_BLAS)) then
print *, "Intel MKL BLAS functions employ threading"
end if
```

`mkl_get_dynamic`

Determines whether Intel MKL is enabled to dynamically change the number of threads.

Syntax

Fortran:

```
ret = mkl_get_dynamic()
```

C:

```
int mkl_get_dynamic(void);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Description

This function returns the status of dynamic adjustment of the number of threads. To determine this status, the function inspects the return value of the following function call and if it is undefined, inspects the environment setting below:

- A call to `mkl_set_dynamic`
- The `MKL_DYNAMIC` environment variable

NOTE

Dynamic adjustment of the number of threads is enabled by default.

The dynamic adjustment works as follows. Suppose that the `mkl_get_max_threads` function returns the number of threads equal to N . If dynamic adjustment is enabled, Intel MKL may request up to N threads, depending on the size of the problem. If dynamic adjustment is disabled, Intel MKL requests exactly N threads for internal parallel regions (provided it uses a threaded algorithm with at least N computations that can be done in parallel). However, the OpenMP* run-time library may be configured to supply fewer threads than Intel MKL requests, depending on the OpenMP* setting of dynamic adjustment.

Return Values

Name	Type	Description
ret	FORTRAN: INTEGER*4 C: int	0 - Dynamic adjustment of the number of threads is disabled. 1 - Dynamic adjustment of the number of threads is enabled.

Example

```
// C/C++
#include "mkl_service.h"
...
int nt = mkl_get_max_threads();
if (1 == mkl_get_dynamic())
printf("Intel MKL may use less than %i threads for a large problem", nt);
else
printf("Intel MKL should use %i threads for a large problem", nt);
```

```

! Fortran
use mkl_service
integer(4) :: nt
...
nt = mkl_get_max_threads()
if (1 == mkl_get_dynamic()) then
    print'("Intel MKL may use less than "I0" threads for a large problem")', nt
else
    print'("Intel MKL should use "I0" threads for a large problem")', nt
end if

```

Error Handling

Error Handling for Linear Algebra Routines

xerbla

Error handling function called by BLAS, LAPACK, VML, and VSL functions.

Syntax

Fortran:

```
call xerbla( srname, info )
```

C:

```
void xerbla( const char * srname, const int* info, const int len );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Input Parameters

Name	Type	Description
<i>srname</i>	FORTRAN: CHARACTER*(*) C: const char*	The name of the routine that called <code>xerbla</code>
<i>info</i>	FORTRAN: INTEGER C: const int*	The position of the invalid parameter in the parameter list of the calling function or an error code
C: <i>len</i>	const int	Length of the source string

Description

The `xerbla` function is an error handler for Intel MKL BLAS, LAPACK, VML, and VSL functions. It is called by a BLAS, LAPACK, VML, or VSL function if an issue is encountered on entry or during the function execution.

`xerbla` operates as follows:

1. Prints a message that depends on the value of the *info* parameter as explained in the [Error Messages Printed by xerbla](#) table.

NOTE

A specific message can differ from the listed messages in numeric values and/or function names.

- 2.** Returns to the calling application.

Comments in the Netlib LAPACK reference code (http://www.netlib.org/lapack/explore-html/d1/dc0/_b_l_a_s_2_s_r_c_2xerbla_8f.html) suggest this behavior although the LAPACK User's Guide recommends that the execution should stop when an error occurs.

Error Messages Printed by xerbla

Value of <i>info</i>	Error Message
1001	Intel MKL ERROR: Incompatible optional parameters on entry to DGEMM.
1212	Intel MKL INTERNAL ERROR: Issue accessing coprocessor in function CGEEV.
1000 or 1089	Intel MKL INTERNAL ERROR: Insufficient workspace available in function CGELSD.
< 0	Intel MKL INTERNAL ERROR: Condition 1 detected in function DLASD8.
Other	Intel MKL ERROR: Parameter 6 was incorrect on entry to DGEMM.

Note that `xerbla` is an internal function. You can change or disable printing of an error message by providing your own `xerbla` function. The following examples illustrate usage of `xerbla`.

Example

Fortran:

```
subroutine xerbla(srname, info)
character*(*) srname !Name of subprogram that called xerbla
integer*4 info !Position of the invalid parameter in the parameter list
return !Return to the calling subprogram end
end
```

C:

```
void xerbla(char* srname, int* info, int len){
// srname - name of the function that called xerbla
// info - position of the invalid parameter in the parameter list
// len - length of the name in bytes
printf("\nXERBLA is called :%s: %d\n",srname,*info);
}
```

See Also

[mkl_set_xerbla](#)

pixerbla

[Error handling routine called by ScaLAPACK routines.](#)

Syntax

```
void pixerbla(MKL_INT* ictxt, char* srname, MKL_INT* info);
```

Include Files

- C: mkl_scalapack.h

Input Parameters

<i>ictxt</i>	(global) MKL_INT
	The BLACS context handle, indicating the global context of the operation. The context itself is global.
<i>srname</i>	(global) char
	The name of the routine which called <code>pixerbla</code> .
<i>info</i>	(global) MKL_INT.
	The position of the invalid parameter in the parameter list of the calling routine.

Description

This routine is an error handler for the *ScalAPACK* routines. It is called if an input parameter has an invalid value. A message is printed and program execution continues. For *ScalAPACK* driver and computational routines, a `RETURN` statement is issued following the call to `pixerbla`.

Control returns to the higher-level calling routine, and you can determine how the program should proceed. However, in the specialized low-level *ScalAPACK* routines (auxiliary routines that are Level 2 equivalents of computational routines), the call to `pixerbla()` is immediately followed by a call to `BLACS_ABORT()` to terminate program execution since recovery from an error at this level in the computation is not possible.

It is always good practice to check for a non-zero value of *info* on return from a *ScalAPACK* routine. Installers may consider modifying this routine in order to call system-specific exception-handling facilities.

Handling Fatal Errors

A fatal error is a circumstance under which Intel MKL cannot continue the computation. For example, a fatal error occurs when Intel MKL cannot load a dynamic library or confronts an unsupported CPU type. In case of a fatal error, the default Intel MKL behavior is to print an explanatory message to the console and call an internal function that terminates the application with a call to the system `exit()` function. Intel MKL enables you to override this behavior by setting a custom handler of fatal errors. The custom error handler can be configured to throw a C++ exception, set a global variable indicating the failure, or otherwise handle cannot-continue situations. It is not necessary for the custom error handler to call the system `exit()` function. Once execution of the error handler completes, a call to Intel MKL returns to the calling program without performing any computations and leaves no memory allocated by Intel MKL and no OpenMP* synchronization pending on return.

To specify a custom fatal error handler, call the `mkl_set_exit_handler` function.

`mkl_set_exit_handler`

Sets the custom handler of fatal errors.

Syntax

Fortran:

```
external :: myexit
interface = mkl_set_exit_handler( myexit )
```

C:

```
int mkl_set_exit_handler (MKLExitHandler myexit);
```

Fortran Include Files/Modules

None.

C Include Files

- mkl.h

Input Parameters

Name	Interface/Prototype	Description
myexit	<p>FORTRAN:</p> <pre>interface subroutine myexit(iwhy) integer,value :: iwhy end subroutine myexit end interface</pre> <p>C:</p> <pre>void (*myexit)(int why);</pre>	The error handler to set.

Description

This function sets the custom handler of fatal errors. If the input parameter is *NULL*, the system `exit()` function is set.

The following example shows how to use a custom handler of fatal errors in your C++ application:

```
#include "mkl.h"
void my_exit(int why){
    throw my_exception();
}
int ComputationFunction()
{
    mkl_set_exit_handler( my_exit );
    try {
        compute_using_mkl();
    }
    catch (const my_exception& e) {
        handle_exception();
    }
}
```

The following example shows how to use a custom handler of fatal errors in your Fortran application:

```
subroutine myexit(rsn)
integer,value :: rsn
call msgbox("Application is terminating")
end myexit

program app
external :: myexit
call mkl_set_exit_handler(myexit)
!.... compute using Intel MKL...
```

Character Equality Testing

Isame

Tests two characters for equality regardless of the case.

Syntax

Fortran:

```
val = lsame( ca, cb )
```

C:

```
int lsame( const char* ca, const char* cb, int lca, int lcb );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Input Parameters

Name	Type	Description
ca, cb	<small>FORTRAN:</small> CHARACTER*1 <small>C:</small> const char*	<small>FORTRAN:</small> The single characters to be compared <small>C:</small> Pointers to the single characters to be compared
C: lca, lcb	int	Lengths of the input character strings, equal to one.

Description

This logical function checks whether two characters are equal regardless of the case.

Return Values

Name	Type	Description
val	<small>FORTRAN:</small> LOGICAL <small>C:</small> int	Result of the comparison: <ul style="list-style-type: none">• .TRUE. in Fortran and a non-zero value in C if ca is the same letter as cb, maybe except for the case.• .FALSE. in Fortran and zero in C if ca and cb are different letters for whatever cases.

Isamen

Tests two character strings for equality regardless of the case.

Syntax

Fortran:

```
val = lsamen( n, ca, cb )
```

C:

```
MKL_INT lsamen( const MKL_INT* n, const char* ca, const char* cb );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	FORTRAN: INTEGER C: const MKL_INT*	FORTRAN: The number of characters in <i>ca</i> and <i>cb</i> to be compared. C: Pointer to the number of characters in <i>ca</i> and <i>cb</i> to be compared.
<i>ca</i> , <i>cb</i>	FORTRAN: CHARACTER* (*) C: const char*	Character strings of length at least <i>n</i> to be compared. Only the first <i>n</i> characters of each string will be accessed.

Description

This logical function tests whether the first *n* letters of one string are the same as the first *n* letters of the other string, regardless of the case.

Return Values

Name	Type	Description
<i>val</i>	FORTRAN: LOGICAL C: MKL_INT	Result of the comparison: <ul style="list-style-type: none"> .TRUE. in Fortran and a non-zero value in C if the first <i>n</i> letters in <i>ca</i> and <i>cb</i> character strings are equal, maybe except for the case, or if the length of character string <i>ca</i> or <i>cb</i> is less than <i>n</i>. .FALSE. in Fortran and zero in C if the first <i>n</i> letters in <i>ca</i> and <i>cb</i> character strings are different for whatever cases.

Timing

second/dsecnd

Returns elapsed time in seconds. Use to estimate real time between two calls to this function.

Syntax

Fortran:

```
val = second()
val = dsecnd()
```

C:

```
float second( void );
double dsecnd( void );
```

Include Files

- Fortran: mkl.fi
- C: mkl.h

Description

The second/dsecnd function returns time in seconds to be used to estimate real time between two calls to the function. The difference between these functions is in the precision of the floating-point type of the result: while second returns the single-precision type, dsecnd returns the double-precision type.

Use these functions to measure durations. To do this, call each of these functions twice. For example, to measure performance of a routine, call the appropriate function directly before a call to the routine to be measured, and then after the call of the routine. The difference between the returned values shows real time spent in the routine.

Initializations may take some time when the second/dsecnd function runs for the first time. To eliminate the effect of this extra time on your measurements, make the first call to second/dsecnd in advance.

Do not use second to measure short time intervals because the single-precision format is not capable of holding sufficient timer precision.

Return Values

Name	Type	Description
val	FORTRAN: REAL for second DOUBLE PRECISION for dsecnd C: float for second double for dsecnd	Elapsed real time in seconds

mkl_get_cpu_clocks

Returns elapsed CPU clocks.

Syntax

Fortran:

```
call mkl_get_cpu_clocks( clocks )
```

C:

```
void mkl_get_cpu_clocks (unsigned MKL_INT64 *clocks);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Output Parameters

Name	Type	Description
<i>clocks</i>	FORTRAN: INTEGER*8 C: unsigned MKL_INT64	Elapsed CPU clocks

Description

The `mkl_get_cpu_clocks` function returns the elapsed CPU clocks.

This may be useful when timing short intervals with high resolution. The `mkl_get_cpu_clocks` function is also applied in pairs like second/dsecnd. Note that out-of-order code execution on IA-32 or Intel® 64 architecture processors may disturb the exact elapsed CPU clocks value a little bit, which may be important while measuring extremely short time intervals.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

`mkl_get_cpu_frequency`

Returns the current CPU frequency value in GHz.

Syntax

Fortran:

```
freq = mkl_get_cpu_frequency()
```

C:

```
double mkl_get_cpu_frequency(void);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Description

The function `mkl_get_cpu_frequency` returns the current CPU frequency in GHz.

NOTE

The returned value may vary from run to run if power management or Intel® Turbo Boost Technology is enabled.

Return Values

Name	Type	Description
<i>freq</i>	FORTRAN: DOUBLE PRECISION C: double	Current CPU frequency value in GHz

mkl_get_max_cpu_frequency

Returns the maximum CPU frequency value in GHz.

Syntax**Fortran:**

```
freq = mkl_get_max_cpu_frequency()
```

C:

```
double mkl_get_max_cpu_frequency(void);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Description

The function `mkl_get_max_cpu_frequency` returns the maximum CPU frequency in GHz.

Return Values

Name	Type	Description
<i>freq</i>	FORTRAN: DOUBLE PRECISION C: double	Maximum CPU frequency value in GHz

mkl_get_clocks_frequency

Returns the frequency value in GHz based on constant-rate Time Stamp Counter.

Syntax**Fortran:**

```
freq = mkl_get_clocks_frequency()
```

C:

```
double mkl_get_clocks_frequency (void);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Description

The function `mkl_get_clocks_frequency` returns the CPU frequency value (in GHz) based on constant-rate Time Stamp Counter (TSC). Use of the constant-rate TSC ensures that each clock tick is constant even if the CPU frequency changes. Therefore, the returned frequency is constant.

NOTE

Obtaining the frequency may take some time when `mkl_get_clocks_frequency` is called for the first time. The same holds for functions `second/dsecnd`, which call `mkl_get_clocks_frequency`.

Return Values

Name	Type	Description
<code>freq</code>	<code>FORTRAN: DOUBLE PRECISION</code> <code>C: double</code>	Frequency value in GHz

See Also

`second/dsecnd`

Using a Fortran Interface Module for Support Functions

Memory Management

This section describes the Intel MKL memory functions. See the *Intel® MKL User's Guide* for more memory usage information.

mkl_free_buffers

Frees unused memory allocated by the Intel MKL Memory Allocator.

Syntax

Fortran:

```
call mkl_free_buffers
```

C:

```
void mkl_free_buffers (void);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Description

To improve performance of Intel MKL, the Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The `mkl_free_buffers` function frees unused memory allocated by the Memory Allocator.

See the *Intel MKL User's Guide* for details.

You should call `mkl_free_buffers` after the last call to Intel MKL functions. In large applications, if you suspect that the memory may get insufficient, you may call this function earlier, but anticipate a drop in performance that may occur due to reallocation of buffers for subsequent calls to Intel MKL functions.

In a threaded application, avoid calling `mkl_free_buffers` from each thread because the function has a global effect. Call `mkl_thread_free_buffers` instead.

Usage of `mkl_free_buffers` with FFT Functions (C Example)

```
DFTI_DESCRIPTOR_HANDLE hand1;
DFTI_DESCRIPTOR_HANDLE hand2;
void mkl_free_buffers(void);
. . .
/* Using MKL FFT */
Status = DftiCreateDescriptor(&hand1, DFTI_SINGLE, DFTI_COMPLEX, dim, m1);
Status = DftiCommitDescriptor(hand1);
Status = DftiComputeForward(hand1, s_array1);
. . .
Status = DftiCreateDescriptor(&hand2, DFTI_SINGLE, DFTI_COMPLEX, dim, m2);
Status = DftiCommitDescriptor(hand2);
. . .
Status = DftiFreeDescriptor(&hand1);
/* Do not call mkl_free_buffers() here because the hand2 descriptor will be corrupted! */
. . .
Status = DftiComputeBackward(hand2, s_array2));
Status = DftiFreeDescriptor(&hand2);
/* Here you finish using Intel MKL FFT */
/* Memory leak will be triggered by any memory control tool */
/* Use mkl_free_buffers() to avoid memory leaking */
mkl_free_buffers();
```

[mkl_thread_free_buffers](#)

Frees unused memory allocated by the Intel MKL Memory Allocator in the current thread.

Syntax

Fortran:

```
call mkl_thread_free_buffers
```

C:

```
void mkl_thread_free_buffers (void);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod

- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Description

To improve performance of Intel MKL, the Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The `mkl_thread_free_buffers` function frees unused memory allocated by the Memory Allocator in the current thread only.

You should call `mkl_thread_free_buffers` after the last call to Intel MKL functions in the current thread. In large applications, if you suspect that the memory may get insufficient, you may call this function earlier, but anticipate a drop in performance that may occur due to reallocation of buffers for subsequent calls to Intel MKL functions.

See Also

[mkl_free_buffers](#)

[Using a Fortran Interface Module for Support Functions](#)

`mkl_disable_fast_mm`

Turns off the Intel MKL Memory Allocator for Intel MKL functions to directly use the system malloc/free functions.

Syntax

Fortran:

```
mm = mkl_disable_fast_mm
```

C:

```
int mkl_disable_fast_mm (void);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Description

The `mkl_disable_fast_mm` function turns the Intel MKL Memory Allocator off for Intel MKL functions to directly use the system `malloc/free` functions. Intel MKL Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The Memory Allocator is turned on by default for better performance. To turn it off, you can use the `mkl_disable_fast_mm` function or the `MKL_DISABLE_FAST_MM` environment variable (See the *Intel MKL User's Guide* for details.) Call `mkl_disable_fast_mm` before calling any Intel MKL functions that require allocation of memory buffers.

NOTE

Turning the Memory Allocator off negatively impacts performance of some Intel MKL routines, especially for small problem sizes.

Return Values

Name	Type	Description
<i>mm</i>	FORTAN: INTEGER*4 C: int	1 - The Memory Allocator is successfully turned off. 0 - Turning the Memory Allocator off failed.

mkl_mem_stat

Reports the status of the Intel MKL Memory Allocator.

Syntax

Fortran:

```
AllocatedBytes = mkl_mem_stat( AllocatedBuffers )
```

C:

```
MKL_INT64 mkl_mem_stat (int* AllocatedBuffers);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Output Parameters

Name	Type	Description
<i>AllocatedBuffers</i>	FORTAN: INTEGER*4 C: int	The number of buffers allocated by Intel MKL.

Description

The function returns the number of buffers allocated by Intel MKL and the amount of memory in these buffers. Intel MKL can allocate the memory buffers internally or in a call to [mkl_malloc/mkl_calloc](#). If no buffers are allocated at the moment, the `mkl_mem_stat` function returns 0. Call `mkl_mem_stat` to check the Intel MKL memory status.

NOTE

If you free all the memory allocated in calls to `mkl_malloc` or `mkl_calloc` and then call `mkl_free_buffers`, a subsequent call to `mkl_mem_stat` normally returns 0.

Return Values

Name	Type	Description
<i>AllocatedBytes</i>	FORTAN: INTEGER*8 C: MKL_INT64	The amount of allocated memory (in bytes).

See Also

[Usage Examples for the Memory Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_peak_mem_usage

Reports the peak memory allocated by the Intel MKL Memory Allocator.

Syntax**Fortran:**

```
AllocatedBytes = mkl_peak_mem_usage( mode )
```

C:

```
MKL_INT64 mkl_peak_mem_usage (int mode);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
mode	FORTRAN: INTEGER*4 C: int	Requested mode of the function's operation. Possible values: <ul style="list-style-type: none"> MKL_PEAK_MEM_ENABLE - start gathering the peak memory data MKL_PEAK_MEM_DISABLE - stop gathering the peak memory data MKL_PEAK_MEM - return the peak memory MKL_PEAK_MEM_RESET - return the peak memory and reset the counter to start gathering the peak memory data from scratch

Description

The `mkl_peak_mem_usage` function reports the peak memory allocated by the Intel MKL Memory Allocator.

Gathering the peak memory data is turned off by default. If you need to know the peak memory, explicitly turn the data gathering mode on by calling the function with the `MKL_PEAK_MEM_ENABLE` value of the parameter. Use the `MKL_PEAK_MEM` and `MKL_PEAK_MEM_RESET` values only when the data gathering mode is turned on. Otherwise the function returns -1. The data gathering mode leads to performance degradation, so when the mode is turned on, you can turn it off by calling the function with the `MKL_PEAK_MEM_DISABLE` value of the parameter.

- If Intel MKL is running in a threaded mode, the `mkl_peak_mem_usage` function may return different amounts of memory from run to run.

- The function reports the peak memory for the entire application, not just for the calling thread.

Return Values

Name	Type	Description
<i>AllocatedBytes</i>	FORTRAN: INTEGER*8 C: MKL_INT64	The peak memory allocated by the Memory Allocator (in bytes) or -1 in case of errors.

See Also

[Usage Examples for the Memory Functions](#)
[Using a Fortran Interface Module for Support Functions](#)

mkl_malloc

Allocates an aligned memory buffer.

Syntax

Fortran:

```
a_ptr = mkl_malloc( alloc_size, alignment )
```

C:

```
void* mkl_malloc (size_t alloc_size, int alignment);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>alloc_size</i>	FORTRAN: INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems C: size_t	Size of the buffer to be allocated.
<i>alignment</i>	FORTRAN: INTEGER*4 C: int	Alignment of the buffer.

Description

The function allocates an *alloc_size*-byte buffer aligned on the *alignment*-byte boundary.

If *alignment* is not a power of 2, the 32-byte alignment is used.

Return Values

Name	Type	Description
<code>a_ptr</code>	FORTRAN: POINTER C: <code>void*</code>	Pointer to the allocated buffer if <code>alloc_size ≥ 1</code> , NULL if <code>alloc_size < 1</code> .

See Also

[mkl_free](#)

[Usage Examples for the Memory Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_calloc

Allocates and initializes an aligned memory buffer.

Syntax

Fortran:

```
a_ptr = mkl_calloc( num, size, alignment )
```

C:

```
void* mkl_calloc (size_t num, size_t size, int alignment);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>num</code>	FORTRAN: INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems C: <code>size_t</code>	The number of elements in the buffer to be allocated.
<code>size</code>	FORTRAN: INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems C: <code>size_t</code>	The size of the element.
<code>alignment</code>	FORTRAN: INTEGER*4 C: int	Alignment of the buffer.

Description

The function allocates a `num*size`-byte buffer, aligned on the `alignment`-byte boundary, and initializes the buffer with zeros.

If *alignment* is not a power of 2, the 64-byte alignment is used.

Return Values

Name	Type	Description
<i>a_ptr</i>	FORTTRAN: POINTER C: void*	Pointer to the allocated buffer if <i>size</i> ≥ 1 , NULL if <i>size</i> < 1.

See Also

[mkl_malloc](#)

[mkl_realloc](#)

[mkl_free](#)

[Usage Examples for the Memory Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_realloc

Changes the size of memory buffer allocated by mkl_malloc/mkl_calloc.

Syntax

Fortran:

```
a_ptr = mkl_realloc( ptr, size )
```

C:

```
void* mkl_realloc (void *ptr, size_t size);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>ptr</i>	FORTTRAN: POINTER C: void*	Pointer to the memory buffer allocated by the <code>mkl_malloc</code> or <code>mkl_calloc</code> function or a NULL pointer.
<i>size</i>	FORTTRAN: INTEGER*4 for 32-bit systems INTEGER*8 for 64-bit systems C: size_t	New size of the buffer.

Description

The function changes the size of the memory buffer allocated by the `mkl_malloc` or `mkl_calloc` function to `size` bytes. The first bytes of the returned buffer up to the minimum of the old and new sizes keep the content of the input buffer. The returned memory buffer can have a different location than the input one. If `ptr` is `NULL`, the function works as `mkl_malloc`.

Return Values

Name	Type	Description
<code>a_ptr</code>	<code>FORTRAN:</code> <code>POINTER</code> <code>C:</code> <code>void*</code>	Pointer to the re-allocated buffer.

See Also

[mkl_malloc](#)
[mkl_calloc](#)
[mkl_free](#)

[Usage Examples for the Memory Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_free

Frees the aligned memory buffer allocated by mkl_malloc/mkl_calloc.

Syntax

Fortran:

```
call mkl_free( a_ptr )
```

C:

```
void mkl_free (void *a_ptr);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>a_ptr</code>	<code>FORTRAN:</code> <code>POINTER</code> <code>C:</code> <code>void*</code>	Pointer to the buffer to be freed

Description

The function frees the buffer pointed by `ptr` and allocated by the `mkl_malloc()` or `mkl_calloc()` function.

See Also

[mkl_malloc](#)

mkl_calloc**Usage Examples for the Memory Functions****Using a Fortran Interface Module for Support Functions****Usage Examples for the Memory Functions****Usage Example in Fortran for 1-dimensional Arrays**

```

PROGRAM FOO
INCLUDE 'mkl.fi'

DOUBLE PRECISION      A,B,C
POINTER      (A_PTR,A(1)), (B_PTR,B(1)), (C_PTR,C(1))
INTEGER       N, I
REAL*8        ALPHA, BETA
INTEGER*8     ALLOCATED_BYTES
INTEGER*8     PEAK_MEMORY
INTEGER*4     ALLOCATED_BUFFERS

#ifndef _SYSTEM_BITS32
    INTEGER*4 MKL_MALLOC, MKL_CALLOC, MKL_REALLOC
    INTEGER*4 ALLOC_SIZE, NUM, SIZE
#else
    INTEGER*8 MKL_MALLOC, MKL_CALLOC, MKL_REALLOC
    INTEGER*8 ALLOC_SIZE, NUM, SIZE
#endif

EXTERNAL      MKL_MALLOC, MKL_FREE, MKL_CALLOC, MKL_REALLOC

ALPHA = 1.1; BETA = -1.2
N = 1000
SIZE = 8
NUM = N*N
ALLOC_SIZE = SIZE*NUM
PEAK_MEMORY = MKL_PEAK_MEM_USAGE(MKL_PEAK_MEM_ENABLE)
A_PTR = MKL_MALLOC(ALLOC_SIZE,64)
B_PTR = MKL_MALLOC(ALLOC_SIZE,64)
C_PTR = MKL_CALLOC(NUM,SIZE,64)
DO I=1,N*N
    A(I) = I
    B(I) = -I
END DO

CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N);

ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
PRINT *, 'DGEMM uses ',ALLOCATED_BYTES,' bytes in ',
$ ALLOCATED_BUFFERS,' buffers '

CALL MKL_FREE_BUFFERS
CALL MKL_FREE(A_PTR)
CALL MKL_FREE(B_PTR)
CALL MKL_FREE(C_PTR)

ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
IF (ALLOCATED_BYTES > 0) THEN
    PRINT *, 'MKL MEMORY LEAK!'
    PRINT *, 'AFTER MKL_FREE_BUFFERS there are ',
$     ALLOCATED_BYTES,' bytes in ',
$     ALLOCATED_BUFFERS,' buffers'

```

```

END IF

PEAK_MEMORY = MKL_PEAK_MEM_USAGE(MKL_PEAK_MEM_RESET)
PRINT *, 'Peak memory allocated by Intel MKL memory allocator ',
$ PEAK_MEMORY, ' bytes. ',
$ 'Start to count new memory peak'

A_PTR = MKL_MALLOC(ALLOC_SIZE,64)
A_PTR = MKL_REALLOC(A_PTR,ALLOC_SIZE*SIZE)
CALL MKL_FREE(A_PTR)
PEAK_MEMORY = MKL_PEAK_MEM_USAGE(MKL_PEAK_MEM)
PRINT *, 'After reset of peak memory counter',
$ 'Peak memory allocated by Intel MKL memory allocator ',
$ PEAK_MEMORY, ' bytes'

STOP
END

```

Usage Example in Fortran for 2-dimensional Arrays

```

PROGRAM FOO
INTEGER N
PARAMETER (N=100)
DOUBLE PRECISION A,B,C
POINTER (A_PTR,A(N,*)), (B_PTR,B(N,*)), (C_PTR,C(N,*))
INTEGER I,J
REAL*8 ALPHA, BETA
INTEGER*8 ALLOCATED_BYTES
INTEGER*4 ALLOCATED_BUFFERS

#ifdef _SYSTEM_BITS32
    INTEGER*4 MKL_MALLOC
    INTEGER*4 ALLOC_SIZE
#else
    INTEGER*8 MKL_MALLOC
    INTEGER*8 ALLOC_SIZE
#endif

INTEGER MKL_MEM_STAT
EXTERNAL MKL_MALLOC, MKL_FREE, MKL_MEM_STAT

ALPHA = 1.1; BETA = -1.2
ALLOC_SIZE = 8*N*N
A_PTR = MKL_MALLOC(ALLOC_SIZE,64)
B_PTR = MKL_MALLOC(ALLOC_SIZE,64)
C_PTR = MKL_MALLOC(ALLOC_SIZE,64)
DO I=1,N
    DO J=1,N
        A(I,J) = I
        B(I,J) = -I
        C(I,J) = 0.0
    END DO
END DO

CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N);

ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
PRINT *, 'DGEMM uses ',ALLOCATED_BYTES,' bytes in ',
$ ALLOCATED_BUFFERS,' buffers '

CALL MKL_FREE_BUFFERS

```

```

CALL MKL_FREE(A_PTR)
CALL MKL_FREE(B_PTR)
CALL MKL_FREE(C_PTR)

ALLOCATED_BYTES = MKL_MEM_STAT(ALLOCATED_BUFFERS)
IF (ALLOCATED_BYTES > 0) THEN
    PRINT *, 'MKL MEMORY LEAK!'
    PRINT *, 'AFTER MKL_FREE_BUFFERS there are ',
$      ALLOCATED_BYTES, ' bytes in ',
$      ALLOCATED_BUFFERS, ' buffers'
END IF

STOP
END

```

Usage Example in C

```

#include <stdio.h>
#include <mkl.h>

int main(void) {
    double *a, *b, *c;
    int n, i;
    double alpha, beta;
    MKL_INT64 AllocatedBytes;
    int N_AllocatedBuffers;

    alpha = 1.1; beta = -1.2;
    n = 1000;
    mkl_peak_mem_usage(MKL_PEAK_MEM_ENABLE);
    a = (double*)mkl_malloc(n*n*sizeof(double), 64);
    b = (double*)mkl_malloc(n*n*sizeof(double), 64);
    c = (double*)mkl_calloc(n*n, sizeof(double), 64);
    for (i=0;i<(n*n);i++) {
        a[i] = (double)(i+1);
        b[i] = (double)(-i-1);
    }

    dgemm("N","N",&n,&n,&n,&alpha,a,&n,b,&n,&beta,c,&n);
    AllocatedBytes = mkl_mem_stat(&N_AllocatedBuffers);
    printf("\nDGEMM uses %d bytes in %d buffers",AllocatedBytes,N_AllocatedBuffers);

    mkl_free_buffers();
    mkl_free(a);
    mkl_free(b);
    mkl_free(c);

    AllocatedBytes = mkl_mem_stat(&N_AllocatedBuffers);
    if (AllocatedBytes > 0) {
        printf("\nMKL memory leak!");
        printf("\nAfter mkl_free_buffers there are %d bytes in %d buffers",
               AllocatedBytes,N_AllocatedBuffers);
    }
    printf("\nPeak memory allocated by Intel MKL memory allocator %d bytes. Start to count new memory
peak",
           mkl_peak_mem_usage(MKL_PEAK_MEM_RESET));
    a = (double*)mkl_malloc(n*n*sizeof(double), 64);
    a = (double*)mkl_realloc(a,2*n*n*sizeof(double));
    mkl_free(a);
    printf("\nPeak memory allocated by Intel MKL memory allocator after reset of peak memory counter %d
bytes\n",

```

```
mkl_peak_mem_usage(MKL_PEAK_MEM)) ;

return 0;
}
```

Single Dynamic Library Control

Intel® MKL provides the Single Dynamic Library (SDL), which enables setting the interface and threading layer for Intel MKL at run time. See *Intel® MKL User's Guide* for details of SDL and layered model concept. This section describes the functions supporting SDL.

mkl_set_interface_layer

Sets the interface layer for Intel MKL at run time. Use with the Single Dynamic Library.

Syntax

Fortran:

```
interface = mkl_set_interface_layer( required_interface )
```

C:

```
int mkl_set_interface_layer (int required_interface);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>required_interface</i>	<small>FORTRAN: INTEGER C: int</small>	<p>Determines the interface layer. Possible values depend on the system architecture. Some of the values are only available on Linux* OS:</p> <ul style="list-style-type: none"> Intel® 64 architecture: <ul style="list-style-type: none"> MKL_INTERFACE_LP64 for the Intel LP64 interface. MKL_INTERFACE_ILP64 for the Intel ILP64 interface. MKL_INTERFACE_LP64+MKL_INTERFACE_GNU for the GNU* LP64 interface on Linux OS. MKL_INTERFACE_ILP64+MKL_INTERFACE_GNU for the GNU ILP64 interface on Linux OS. IA-32 architecture: <ul style="list-style-type: none"> MKL_INTERFACE_LP64 for the Intel interface on Linux OS. MKL_INTERFACE_LP64+MKL_INTERFACE_GNU or

Name	Type	Description
		MKL_INTERFACE_GNU for the GNU interface on Linux OS.

Description

If you are using the Single Dynamic Library (SDL), the `mkl_set_interface_layer` function sets the specified interface layer for Intel MKL at run time.

Call this function prior to calling any other Intel MKL function in your application except `mkl_set_threading_layer`. You can call `mkl_set_interface_layer` and `mkl_set_threading_layer` in any order.

The `mkl_set_interface_layer` function takes precedence over the `MKL_INTERFACE_LAYER` environment variable.

See *Intel MKL User's Guide* for the layered model concept and usage details of the SDL.

Return Values

Type	Description
<code>FORTRAN: INTEGER</code>	<ul style="list-style-type: none"> Current interface layer if it is set in a call to <code>mkl_set_interface_layer</code> or specified by environment variables or defaults.
<code>C: int</code>	<ul style="list-style-type: none"> Possible values are specified in Input Parameters. -1, if the layer was not specified prior to the call and the input parameter is incorrect.

`mkl_set_threading_layer`

Sets the threading layer for Intel MKL at run time. Use with the Single Dynamic Library (SDL).

Syntax

Fortran:

```
threading = mkl_set_threading_layer( required_threading )
```

C:

```
int mkl_set_threading_layer (int required_threading);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<code>required_threading</code>	<code>FORTRAN:</code> INTEGER <code>C:</code> int	Determines the threading layer. Possible values: <code>MKL_THREADING_INTEL</code> for Intel threading. <code>MKL_THREADING_SEQUENTIAL</code> for the sequential mode of Intel MKL. <code>MKL_THREADING_PGI</code> for PGI threading on Windows* or Linux* operating system only. Do not use this value with the SDL for Intel® Many Integrated Core (Intel® MIC) Architecture. <code>MKL_THREADING_GNU</code> for GNU threading on Linux* operating system only. Do not use this value with the SDL for Intel MIC Architecture.

Description

If you are using the Single Dynamic Library (SDL), the `mkl_set_threading_layer` function sets the specified threading layer for Intel MKL at run time.

Call this function prior to calling any other Intel MKL function in your application except `mkl_set_interface_layer`.

You can call `mkl_set_threading_layer` and `mkl_set_interface_layer` in any order.

The `mkl_set_threading_layer` function takes precedence over the `MKL_THREADING_LAYER` environment variable.

See *Intel MKL User's Guide* for the layered model concept and usage details of the SDL.

Return Values

Type	Description
<code>FORTRAN:</code> INTEGER <code>C:</code> int	<ul style="list-style-type: none"> Current threading layer if it is set in a call to <code>mkl_set_threading_layer</code> or specified by environment variables or defaults. Possible values are specified in Input Parameters. -1, if the layer was not specified prior to the call and the input parameter is incorrect.

`mkl_set_xerbla`

Replaces the error handling routine. Use with the Single Dynamic Library on Windows OS.*

Syntax

Fortran:

```
old_xerbla_ptr = mkl_set_xerbla( new_xerbla_ptr )
```

C:

```
XerblaEntry mkl_set_xerbla (XerblaEntry new_xerbla_ptr);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`

- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<code>new_xerbla_ptr</code>	XerblaEntry	Pointer to the error handling routine to be used.

Description

If you are linking with the Single Dynamic Library (SDL) `mkl_rt.lib` on Windows* OS, the `mkl_set_xerbla` function replaces the error handling routine that is called by Intel MKL functions with the routine specified by the parameter.

See *Intel MKL User's Guide* for details of SDL.

Return Values

The function returns the pointer to the replaced error handling routine.

See Also

`xerbla`

[Using a Fortran Interface Module for Support Functions](#)

`mkl_set_progress`

Replaces the progress information routine. Use with the Single Dynamic Library (SDL) on Windows OS.*

Syntax

Fortran:

```
old_progress_ptr mkl_set_progress( new_progress_ptr )
```

C:

```
ProgressEntry mkl_set_progress (ProgressEntry new_progress_ptr);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<code>new_progress_ptr</code>	ProgressEntry	Pointer to the progress information routine to be used.

Description

If you are linking with the Single Dynamic Library (SDL) `mkl_rt.lib` on Windows* OS, the `mkl_set_progress` function replaces the currently used progress information routine with the routine specified by the parameter.

See *Intel MKL User's Guide* for details of SDL.

Return Values

The function returns the pointer to the replaced progress information routine.

See Also

`mkl_progress`

Using a Fortran Interface Module for Support Functions

Intel Many Integrated Core Architecture Support

Intel MKL functions to support the use of Intel Xeon Phi coprocessors are as follows:

Fortran

```
rc = mkl_mic_enable( )
rc = mkl_get_device_count( )
rc = mkl_mic_set_workdivision( MKL_TARGET_MIC, MKL_MIC_DEFAULT_TARGET_NUMBER, 0.0 )
rc = mkl_mic_set_workdivision( MKL_TARGET_MIC, 0, MKL_MIC_AUTO_WORKDIVISION )
rc = mkl_mic_get_workdivision( MKL_TARGET_MIC, 0, wd )
rc = mkl_mic_set_max_memory( MKL_TARGET_MIC, 0, mem_size)
rc = mkl_mic_set_device_num_threads( MKL_TARGET_MIC, 0, num_threads)
rc = mkl_mic_set_resource_limit( fraction )
rc = mkl_mic_get_resource_limit( fraction )
rc = mkl_mic_set_offload_report( 1 )
rc = mkl_mic_free_memory( MKL_TARGET_MIC, 0 )
rc = mkl_mic_disable( )
```

C

```
#include "mkl.h" // Required for Intel MIC Architecture functions
rc = mkl_mic_enable();
rc = mkl_get_device_count();
rc = mkl_mic_set_workdivision(MKL_TARGET_MIC, MKL_MIC_DEFAULT_TARGET_NUMBER, 0.0);
rc = mkl_mic_set_workdivision(MKL_TARGET_MIC, 0, MKL_MIC_AUTO_WORKDIVISION);
rc = mkl_mic_get_workdivision(MKL_TARGET_MIC, 0, &wd);
rc = mkl_mic_set_max_memory(MKL_TARGET_MIC, 0, mem_size);
rc = mkl_mic_set_device_num_threads(MKL_TARGET_MIC, 0, num_threads);
rc = mkl_mic_set_resource_limit(fraction);
rc = mkl_mic_get_resource_limit(fraction);
rc = mkl_mic_set_offload_report(1);
rc = mkl_mic_free_memory(MKL_TARGET_MIC, 0);
rc = mkl_mic_disable();
```

The `mkl_mic_enable` function enables Intel MKL to offload computations to Intel Xeon Phi coprocessors automatically, while the `mkl_mic_disable` function disables automatic offloading.

Optional work-division control functions enable you to specify the fractional amount of work to distribute between the host CPU and the coprocessors. Work division is a fractional measure ranging from 0.0 to 1.0. For example, setting work division for the host CPU to 0.5 means to keep half of the computational work on the host CPU and move half to the coprocessor(s). Setting work division to 0.25 for a coprocessor means to offload a quarter of the computational work to this coprocessor while leaving the rest on the host CPU.

Other functions enable more control over computational resources of Intel Xeon Phi coprocessors.

NOTE

The support functions for Intel MIC Architecture take precedence over the respective environment variables.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

mkl_mic_enable

Enables Automatic Offload mode.

Syntax**Fortran:**

```
ierr = mkl_mic_enable( )
```

C:

```
int mkl_mic_enable();
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Description

The `mkl_mic_enable` function enables Automatic Offload mode and initializes Intel Xeon Phi coprocessors available on your system. This function takes precedence over the `MKL_MIC_ENABLE` environment variable. Unlike the function, the environment variable enables Automatic Offload mode, but the initialization is delayed until a call to an Intel MKL function supporting Automatic Offload. For functions supporting Automatic Offload, see the *Intel MKL Release Notes*.

Return Values

Name	Type	Description
<code>ierr</code>	FORTRAN: INTEGER*4 C: int	Result status: = 0 Indicates that Automatic Offload mode is successfully enabled. < 0 Indicates a failure to enable Automatic Offload mode.

mkl_mic_disable

Disables Automatic Offload mode.

Syntax

Fortran:

```
ierr = mkl_mic_disable( )
```

C:

```
int mkl_mic_disable();
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Description

The `mkl_mic_disable` function disables Automatic Offload mode. To enable Automatic Offload mode back, use the `mkl_mic_enable` function.

NOTE

If Automatic Offload mode is not enabled, a call to `mkl_mic_disable` completes successfully if Intel Xeon Phi coprocessors are available on your system.

Return Values

Name	Type	Description
<i>ierr</i>	FORTRAN: INTEGER*4 C: int	Result status: = 0 Indicates that Automatic Offload mode is successfully disabled. < 0 Indicates a failure to disable Automatic Offload mode.

mkl_mic_get_device_count

Returns the number of Intel Xeon Phi coprocessors on the system when called on the host CPU.

Syntax

Fortran:

```
ndevices = mkl_mic_get_device_count( )
```

C:

```
int mkl_mic_get_device_count();
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Description

The `mkl_get_device_count` function returns the number of Intel Xeon Phi coprocessors on your system. You may need this number to specify a custom work-division scheme.

CAUTION

Call this function only on the host CPU.

Return Values

Name	Type	Description
<code>ndevices</code>	<code>FORTRAN:</code> INTEGER*4 <code>C:</code> int	The number of Intel Xeon Phi coprocessors available on the system. Equals zero if there are none.

See Also

[mkl_mic_enable](#)
[Using a Fortran Interface Module for Support Functions](#)

[mkl_mic_set_workdivision](#)

For computations in the Automatic Offload mode, sets the fraction of the work for the specified coprocessor or host CPU to do.

Syntax

Fortran:

```
ierr = mkl_mic_set_workdivision( target_type, target_number, wd )
```

C:

```
int mkl_mic_set_workdivision (MKL_MIC_TARGET_TYPE target_type, int target_number,  
double wd);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<code>target_type</code>	<code>FORTRAN</code> : INTEGER*4 <code>C</code> : MKL_MIC_TARGET_TYPE	Type of the target device. Use one of the following values: <ul style="list-style-type: none">• <code>MKL_TARGET_HOST</code> - host CPU• <code>MKL_TARGET_MIC</code> - Intel Xeon Phi coprocessor, default
<code>target_number</code>	<code>FORTRAN</code> : INTEGER*4 <code>C</code> : int	The device to set the fraction of work for. Takes the following values: <ul style="list-style-type: none">• ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <code>target_number</code> modulo the number of Intel Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>. For example: for a system with 4 Intel Xeon Phi coprocessors, <code>target_number = 6</code> determines the coprocessor number 2.• <0. Reserved. If <code>target_type = MKL_TARGET_HOST</code> , the function ignores the <code>target_number</code> parameter, which may have any value.
<code>wd</code>	<code>FORTRAN</code> : REAL*8 <code>C</code> : double	The fractional amount of the work that the specified device should do, where $0.0 \leq wd \leq 1.0$. Specifying <code>MKL_MIC_AUTO_WORKDIVISION</code> for <code>wd</code> indicates that Intel MKL should determine the amount of the work for the specified device.

Description

If you are using Intel MKL in Automatic Offload mode, the `mkl_mic_set_workdivision` function specifies how much work each Intel Xeon Phi coprocessor or the host CPU should do. This function takes precedence over the `MKL_HOST_WORKDIVISION`, `MKL_MIC_WORKDIVISION`, and `MKL_MIC_<number>_WORKDIVISION` environment variables (see the *Intel MKL User's Guide* for details).

-
- Intel MKL interprets the fraction of work set by the `mkl_mic_set_workdivision` function as guidance toward dividing work between coprocessors, but the library may choose a different work division if necessary.
 - Intel MKL resolves the collision that arises if the sum of all the work-division fractions does not equal one.
 - For LAPACK routines, setting the fraction of work to any value other than 0.0 enables the specified processor for Automatic Offload mode. However Intel MKL LAPACK does not use the value specified to divide the workload. For example, setting the fraction to 0.5 has the same effect as setting the fraction to 1.0.
-

Return Values

Name	Type	Description
<i>ierr</i>	FORTRAN: INTEGER*4 C: int	Result status: = 0 Indicates that the fraction is successfully set. < 0 Indicates a failure to set the fraction.

See Also

[mkl_mic_get_device_count](#)

[mkl_mic_enable](#)

[mkl_mic_get_workdivision](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_get_workdivision

For computations in the Automatic Offload mode, retrieves the fraction of the work for the specified coprocessor or host CPU to do.

Syntax

Fortran:

```
ierr = mkl_mic_get_workdivision( target_type, target_number, wd )
```

C:

```
int mkl_mic_get_workdivision (MKL_MIC_TARGET_TYPE target_type, int target_number,  
double *wd);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>target_type</i>	FORTRAN: INTEGER*4 C: MKL_MIC_TARGET_TYPE	Type of the target device. Use one of the following values: <ul style="list-style-type: none"> MKL_TARGET_HOST - host CPU MKL_TARGET_MIC - Intel Xeon Phi coprocessor, default
<i>target_number</i>	FORTRAN: INTEGER*4 C: int	The device to retrieve the fraction of work for. Takes the following values: <ul style="list-style-type: none"> ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <i>target_number</i> modulo the number of Intel

Name	Type	Description
		<p>Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>. For example: for a system with 4 Intel Xeon Phi coprocessors, <code>target_number</code> = 6 determines the coprocessor number 2.</p> <ul style="list-style-type: none"> • <0. Reserved. <p>If <code>target_type</code> = <code>MKL_TARGET_HOST</code>, the function ignores the <code>target_number</code> parameter, which may have any value.</p>

Output Parameters

Name	Type	Description
<code>wd</code>	FORTRAN: <code>REAL*8</code> C: <code>double</code>	<p>The fractional amount of the work that the specified device should do, where $0.0 \leq wd \leq 1.0$.</p> <p><code>MKL_MIC_AUTO_WORKDIVISION</code> set for <code>wd</code> indicates that Intel MKL should determine the amount of the work for the specified device.</p>

Description

If you are using Intel MKL in the Automatic Offload mode, the `mkl_mic_get_workdivision` function provides you with the amount of the work that the specified coprocessor or host CPU is configured to do.

Return Values

Name	Type	Description
<code>ierr</code>	FORTRAN: <code>INTEGER*4</code> C: <code>int</code>	<p>Result status:</p> <ul style="list-style-type: none"> = 0 Indicates that the fraction is successfully returned. < 0 Indicates a failure to retrieve the fraction.

See Also

[mkl_mic_get_device_count](#)

[mkl_mic_enable](#)

[mkl_mic_set_workdivision](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_set_max_memory

Sets the maximum amount of Intel Xeon Phi coprocessor memory reserved for the Automatic Offload computations.

Syntax

Fortran:

```
ierr = mkl_mic_set_max_memory( target_type, target_number, mem_size )
```

C:

```
int mkl_mic_set_max_memory (MKL_MIC_TARGET_TYPE target_type, int target_number, size_t
mem_size);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>target_type</i>	FORTRAN: INTEGER*4 C: MKL_MIC_TARGET_TYPE	Type of the target device. Use the value of MKL_TARGET_MIC - Intel Xeon Phi coprocessor, default.
<i>target_number</i>	FORTRAN: INTEGER*4 C: int	The coprocessor number for which the maximum memory used for Automatic Offload computations is set. Takes the following values: <ul style="list-style-type: none"> • ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <i>target_number</i> modulo the number of Intel Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>. For example: for a system with 4 Intel Xeon Phi coprocessors, <i>target_number</i> = 6 determines the coprocessor number 2. • <0. Reserved.
<i>mem_size</i>	FORTRAN: INTEGER*8 C: size_t	For the <i>target_number</i> device, the amount of memory in kilobytes to reserve for Automatic Offload computations. Intel MKL attempts to not exceed the specified memory size for Automatic Offload computations.

Description

The `mkl_mic_set_max_memory` function enables you to limit coprocessor memory used by the Automatic Offload computations. Intel MKL reserves the specified memory on Intel Xeon Phi coprocessors. This can improve the performance of Automatic Offload computations by reducing the cost of buffer initialization and data transfer. The specified memory is reserved for the calling process, and the threads of a process share the specified memory. Intel MKL allocates additional memory for each process that performs Automatic Offload computations.

This function takes precedence over the `MKL_MIC_MAX_MEMORY` and `MKL_MIC_<number>_MAX_MEMORY` environment variables (see the *Intel MKL User's Guide* for details).

-
- Call `mkl_mic_set_max_memory` before any Intel MKL functions that do Automatic Offload computations.
 - Use `mkl_mic_free_memory` to free the coprocessor memory reserved for the Automatic Offload computations.
-

Return Values

Name	Type	Description
<i>ierr</i>	FORTRAN: INTEGER*4 C: int	Result status: = 0 Indicates the coprocessor memory reserved for Automatic Offload is set successfully. < 0 Indicates an error.

See Also

[mkl_mic_free_memory](#)

[mkl_mic_enable](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_free_memory

Frees the coprocessor memory reserved for the Automatic Offload computations.

Syntax

Fortran:

```
ierr = mkl_mic_free_memory( target_type, target_number )
```

C:

```
int mkl_mic_free_memory (MKL_MIC_TARGET_TYPE target_type, int target_number);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>target_type</i>	FORTRAN: INTEGER*4 C: MKL_MIC_TARGET_TYPE	Type of the target device. Use the value of MKL_TARGET_MIC - Intel Xeon Phi coprocessor, default.
<i>target_number</i>	FORTRAN: INTEGER*4 C: int	The coprocessor number for which the memory reserved for Automatic Offload computations is freed. Takes the following values: <ul style="list-style-type: none"> ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <i>target_number</i> modulo the number of Intel Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>.

Name	Type	Description
		<p>For example: for a system with 4 Intel Xeon Phi coprocessors, <code>target_number = 6</code> determines the coprocessor number 2.</p> <ul style="list-style-type: none"> • <0. Reserved.

Description

The `mkl_mic_free_memory` function frees the coprocessor memory reserved for the Automatic Offload computations. If you call `mkl_mic_set_max_memory` to specify the maximum coprocessor memory for Automatic Offload computations, Intel MKL reserves and reuses the specified coprocessor memory during multiple Automatic Offload calls. You can reclaim the coprocessor memory by calling `mkl_mic_free_memory`.

-
- Currently, Intel MKL reserves the coprocessor memory only if the `mkl_mic_set_max_memory` function is called. Therefore, `mkl_mic_free_memory` has no effect unless there is a prior call to the `mkl_mic_set_max_memory` function.
 - If you do not call `mkl_mic_free_memory`, Intel MKL frees the coprocessor memory at the program exit.
-

Return Values

Name	Type	Description
<code>ierr</code>	<code>FORTRAN: INTEGER*4</code> <code>C: int</code>	<p>Result status:</p> <p>= 0 Indicates the coprocessor memory reserved for Automatic Offload is freed successfully.</p> <p>< 0 Indicates an error.</p>

See Also

[mkl_mic_set_max_memory](#)

[mkl_mic_enable](#)

[Using a Fortran Interface Module for Support Functions](#)

[mkl_mic_register_memory](#)

Enables/disables the `mkl_malloc` function running in Automatic Offload mode to register allocated memory.

Syntax

Fortran:

```
call mkl_mic_register_memory( control )
```

C:

```
void mkl_mic_register_memory (int control);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<code>control</code>	<code>FORTAN: INTEGER*4</code> <code>C: int</code>	Desired behavior of <code>mkl_malloc</code> . Possible values: 0 - not register allocated memory, default. 1 - register allocated memory if Automatic Offload (AO) mode is enabled.

Description

The `mkl_mic_register_memory` function enables or disables the `mkl_malloc` function to register allocated memory when `mkl_malloc` runs in AO mode.

Registration of memory may reduce the overhead introduced by the operating system during data transfers between Intel Xeon Phi coprocessors and the host CPU.

This function takes precedence over the `MKL_MIC_REGISTER_MEMORY` environment variable.

If AO mode is disabled, the function has no effect.

See Also

`mkl_malloc` Allocates an aligned memory buffer.

`mkl_mic_enable` Enables Automatic Offload mode.

[Using a Fortran Interface Module for Support Functions](#)

`mkl_mic_set_device_num_threads`

Sets the maximum number of threads to use on an Intel Xeon Phi coprocessor for the Automatic Offload computations.

Syntax

Fortran:

```
ierr = mkl_mic_set_device_num_threads( target_type, target_number, num_threads )
```

C:

```
int mkl_mic_set_device_num_threads (MKL_MIC_TARGET_TYPE target_type, int target_number,
int num_threads);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>target_type</i>	FORTRAN: INTEGER*4 C: MKL_MIC_TARGET_TYPE	Type of the target device. Use the value of MKL_TARGET_MIC - Intel Xeon Phi coprocessor, default.
<i>target_number</i>	FORTRAN: INTEGER*4 C: int	The coprocessor number for which the maximum number of threads to be used for Automatic Offload computations is set. Takes the following values: <ul style="list-style-type: none">• ≥ 0. Specifies execution on a specific coprocessor. The coprocessor is determined by <i>target_number</i> modulo the number of Intel Xeon Phi coprocessors on the system as returned by <code>mkl_mic_get_device_count()</code>. For example: for a system with 4 Intel Xeon Phi coprocessors, <i>target_number</i> = 6 determines the coprocessor number 2.• < 0. Reserved.
<i>num_threads</i>	FORTRAN: INTEGER*4 C: int	The number of threads to use for Automatic Offload computations on the <i>target_number</i> coprocessor. Must be greater than 0.

Description

The `mkl_mic_set_device_num_threads` function enables you to limit the number of threads to use for Automatic Offload computations on a specific Intel Xeon Phi coprocessor.

This function takes precedence over the `MIC_OMP_NUM_THREADS` and `MKL_MIC_<number>_OMP_NUM_THREADS` environment variables (see the *Intel MKL User's Guide* for details).

Unless the maximum number of threads is set by the environment variables or this function, Intel MKL uses all coprocessor cores.

Call `mkl_mic_set_device_num_threads` before initialization of Automatic Offload mode. Otherwise the function returns -1 without setting the number of threads.

NOTE

Automatic Offload mode can be initialized as follows:

- *explicitly*, in a call to the `mkl_mic_enable` function.
- *implicitly*, in the first call to an Intel MKL function that does Automatic Offload computations (for example: ?GEMM). See the *Intel MKL Release Notes* for which functions support Automatic Offload mode.

The `mkl_mic_set_device_num_threads` function sets the maximum number of threads to use for the Automatic Offload computations on Intel Xeon Phi coprocessors only. To control host CPU threading, use general threading control functions and see "Using Additional Threading Control" in the *Intel MKL User's Guide* for more information.

Return Values

Name	Type	Description
<i>ierr</i>	FORTRAN: INTEGER*4 C: int	Result status: = 0 Indicates that the number of threads is set successfully. < 0 Indicates a failure to set the number of threads.

See Also

[mkl_mic_enable](#)

[Using a Fortran Interface Module for Support Functions](#)

[Threading Control](#)

mkl_mic_set_resource_limit

For computations in the Automatic Offload mode, sets the maximum fraction of available Intel Xeon Phi coprocessor computational resources (cores) that the calling process can use.

Syntax

Fortran:

```
ierr = mkl_mic_set_resource_limit( fraction )
```

C:

```
int mkl_mic_set_resource_limit (double fraction);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>fraction</i>	FORTRAN: REAL*8 C: double	The fractional amount of Intel Xeon Phi coprocessor computational resources that the calling process can use for Automatic Offload (AO). Possible values: $0.0 \leq \text{fraction} \leq 1.0$. Special values: <ul style="list-style-type: none"> 0.0 - you need to manage the coprocessors explicitly using the <code>mkl_mic_set_device_num_threads</code> function or the <code>MIC_KMP_AFFINITY</code> environment variable. Default.

Name	Type	Description
		<ul style="list-style-type: none"> • <code>MKL_MPI_PPN</code> constant - enables a fully automated mode for message-passing interface (MPI) applications. In this mode, Intel MKL tries to read the number of MPI processes per node (<i>ppn</i>) from the environment variables passed to the process by MPI. If this attempt is successful, Intel MKL sets the actual value of the fractional amount to $1.0/ppn$.
NOTE		
For Intel® MPI Library, Open MPI, and IBM Platform MPI, Intel MKL automatically detects <i>ppn</i> . For other MPI implementations, use the <code>MKL_MPI_PPN</code> environment variable to set <i>ppn</i> .		

Description

If you are using Intel MKL in the AO mode, the `mkl_mic_set_resource_limit` function specifies how much of the computational resources of Intel Xeon Phi coprocessors can be used by the calling process. Use this function if you need to share coprocessor cores automatically across multiple processes that call Intel MKL in the AO mode. For example, this might be useful in MPI applications.

You can also enable this functionality using the `MKL_MIC_RESOURCE_LIMIT` environment variable (see the *Intel MKL User's Guide* for details), but the `mkl_mic_set_resource_limit` function take precedence over the environment variable.

If *fraction* is set to a valid non-zero value, Intel MKL enables automatic reservation of Intel Xeon Phi coprocessor cores.

Actual reservation is made during a call to an Intel MKL AO function and works as follows:

1. Intel MKL converts *fraction* to a number of Intel Xeon Phi coprocessor cores and tries to find cores that are not reserved by other processes.

The number of available cores can be less than the requested number. Intel MKL considers cores of all available coprocessors as a single space and can find cores on different coprocessors. However, Intel MKL tends to reserve as many cores as were requested and to reserve cores compactly (using as few coprocessors as possible).

2. Intel MKL exclusively reserves cores found in the previous step.

Intel MKL makes reservation safely across a persistent shared memory region using semaphores.

3. Intel MKL performs AO computations on reserved cores only.

You do not need to set any threading parameters for Intel Xeon Phi coprocessors, such as `MIC_OMP_NUM_THREADS` or `MIC_KMP_AFFINITY` (these settings are ignored when automatic resource sharing is enabled).

4. Upon completion of an AO call, Intel MKL releases reserved cores.

Be aware of the following features of the automatic reservation:

- Execution of Intel MKL AO functions falls back to the host if Intel MKL fails to reserve the minimum number of Intel Xeon Phi coprocessor cores that provide a benefit over executing on the host.
- The last core of Intel Xeon Phi coprocessors cannot be reserved because it is used for system processes.
- Reservation works only within a space of a single user. In other words, Intel MKL cannot share Intel Xeon Phi coprocessor resources across processes from different users.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Examples

1. One Intel Xeon Phi coprocessor with 61 cores is available on the system. One process calls `mkl_mic_set_resource_limit(1.0)` and then calls `dgemm` in AO mode. As a result, 60 Intel Xeon Phi coprocessor cores are reserved at the beginning of AO `dgemm`. Then `mkl_mic_set_resource_limit(0.3)` is invoked and AO `dgemm` is called one more time. As a result, `dgemm` is run on 18 Intel Xeon Phi coprocessor cores. During this run of AO `dgemm`, another process calls `mkl_mic_set_resource_limit(1.0)` and invokes AO `dgetrf`. Because 18 cores are reserved by the first process, only 42 cores are reserved for AO `dgetrf`.
2. Two Intel Xeon Phi coprocessors with 61 cores each are available on the system. Three processes simultaneously call `mkl_mic_set_resource_limit(0.34)` and then call `dpotrf` in AO mode. As a result, one process receives 40 cores from coprocessor 1, another process receives 40 cores from coprocessor 2, and the remaining process is given 20 cores from each of the two coprocessors for a total of 40 cores.
3. Two Intel Xeon Phi coprocessors with 58 cores each are available on the system. The user sets `MKL_MIC_PPN=4` and runs an MPI application with 4 MPI ranks. The following sequence of functions is called in each MPI process:

```
mkl_mic_set_workdivision(MKL_TARGET_MIC, 2, 0.0);
mkl_mic_set_resource_limit(MKL_MPI_PPN);
MPI_Init(...);
//calls to Intel MKL AO functions;
MPI_Finalize(...);
```

As a result, actual *fraction* is set to 0.25 for each MPI process, and each MPI process receives 14 cores on Intel Xeon Phi coprocessor 1 (because the user excluded co-processor 2 from computations by setting zero workdivision for it).

Return Values

Name	Type	Description
<i>ierr</i>	FORTRAN: INTEGER*4 C: int	Result status: = 0 Indicates that the fraction is set successfully. < 0 Indicates a failure to set the fraction.

See Also

[mkl_mic_get_resource_limit](#)

[mkl_mic_enable](#)

[mkl_mic_get_device_count](#)

[Using a Fortran Interface Module for Support Functions](#)

[mkl_mic_get_resource_limit](#)

For computations in the Automatic Offload mode, retrieves the maximum fraction of available Intel Xeon Phi coprocessor computational resources (cores) that the calling process can use.

Syntax

Fortran:

```
ierr = mkl_mic_get_resource_limit( fraction )
```

C:

```
int mkl_mic_get_resource_limit (double* fraction);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Output Parameters

Name	Type	Description
<i>fraction</i>	FORTRAN: REAL*8 C: double*	The fractional amount of Intel Xeon Phi coprocessor computational resources that the calling process can use for Automatic Offload (AO). Possible values: $0.0 \leq \text{fraction} \leq 1.0$.

Description

If you are using Intel MKL in the AO mode, the `mkl_mic_get_resource_limit` function retrieves the fractional amount of Intel Xeon Phi coprocessor computational resources that can be used by the calling process.

Return Values

Name	Type	Description
<i>ierr</i>	FORTRAN: INTEGER*4 C: int	Result status: = 0 Indicates that the fraction is successfully returned. < 0 Indicates a failure to return the fraction.

See Also

[mkl_mic_set_resource_limit](#)

[mkl_mic_enable](#)

[mkl_mic_get_device_count](#)

[mkl_mic_get_workdivision](#)

[Using a Fortran Interface Module for Support Functions](#)

mkl_mic_set_offload_report

Turns on/off reporting of Automatic Offload profiling.

Syntax

Fortran:

```
iprev = mkl_mic_set_offload_report( enabled )
```

C:

```
int mkl_mic_set_offload_report (int enabled);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
enabled	FORTRAN: INTEGER*4 C: int	This parameter specifies whether to turn the reporting on or off. Takes values with boolean semantics: <ul style="list-style-type: none"> • 0 - Reporting of Automatic Offload profiling should be turned off. • 1 - Reporting of Automatic Offload profiling should be turned on.

Description

If the OFFLOAD_REPORT environment variable has the value of 1 or 2, the `mkl_mic_set_offload_report` function turns on/off reporting of the Automatic Offload profiling at run time. If the OFFLOAD_REPORT environment variable is not set or is set to a value different from 1 and 2, the function has no effect. If OFFLOAD_REPORT is set to 1 or 2, the reporting is turned on at a program startup. The `mkl_mic_set_offload_report` function does not change the reporting level or the value of the environment variable. Instead, it determines which Intel MKL functions called in the Automatic Offload mode produce profiling reports.

For details of the reporting level and OFFLOAD_REPORT environment variable, see the *Intel MKL User's Guide*.

Return Values

Name	Type	Description
iprev	FORTRAN: INTEGER*4 C: int	The previous value that determined whether the reporting was on or off.

Conditional Numerical Reproducibility Control

The CNR mode of Intel MKL ensures bitwise reproducible results from run to run of Intel MKL functions on a fixed number of threads for a specific Intel instruction set architecture (ISA) under the following conditions:

- Calls to Intel MKL occur in a single executable
- The number of computational threads used by the library does not change in the run

Intel MKL offers both functions and environment variables to support conditional numerical reproducibility. See the *Intel MKL User's Guide* for more information on bitwise reproducible results of computations and for details about the environment variables.

The support functions enable you to configure the CNR mode and also provide information on the current and optimal CNR branch on your system. [Usage Examples for CNR Support Functions](#) illustrate usage of these functions.

Important

Call the functions that define the behavior of CNR before any of the math library functions that they control.

Intel MKL provides named constants for use as input and output parameters of the functions instead of integer values. See [Named Constants for CNR Control](#) for a list of the named constants.

Although you can configure the CNR mode using either the support functions or the environment variables, the functions offer more flexible configuration and control than the environment variables. Settings specified by the functions take precedence over the settings specified by the environment variables.

Use Intel MKL in the CNR mode only in case a need for bitwise reproducible results is critical. Otherwise, run Intel MKL as usual to avoid performance degradation.

While you can supply unaligned input and output data to Intel MKL functions running in the CNR mode, use of aligned data is recommended. Refer to [Reproducibility Conditions](#) for more details.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

[**mkl_cbwr_set**](#)

Configures the CNR mode of Intel MKL.

Syntax

Fortran:

```
status = mkl_cbwr_set( setting )
```

C:

```
int mkl_cbwr_set (int setting);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>setting</i>	FORTRAN: INTEGER*4 C: int	CNR branch to set. See Named Constants for CNR Control for a list of named constants that specify the settings.

Description

The `mkl_cbwr_set` function configures the CNR mode. In this release, it sets the CNR branch and turns on the CNR mode.

NOTE

Settings specified by the `mkl_cbwr_set` function take precedence over the settings specified by the `MKL_CBWR` environment variable.

Return Values

Name	Type	Description
<i>status</i>	FORTRAN: INTEGER*4 C: int	<p>The status of the function completion:</p> <ul style="list-style-type: none"> • <code>MKL_CBWR_SUCCESS</code> - the function completed successfully. • <code>MKL_CBWR_ERR_INVALID_INPUT</code> - an invalid setting is requested. • <code>MKL_CBWR_ERR_UNSUPPORTED_BRANCH</code> - the input value of the branch does not match the instruction set architecture (ISA) of your system. See Named Constants for CNR Control for more details. • <code>MKL_CBWR_ERR_MODE_CHANGE_FAILURE</code> - the <code>mkl_cbwr_set</code> function requested to change the current CNR branch after a call to some Intel MKL function other than a CNR function.

See Also

[Usage Examples for CNR Support Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

`mkl_cbwr_get`

Returns the current CNR settings.

Syntax

Fortran:

```
setting = mkl_cbwr_get( option )
```

C:

```
int mkl_cbwr_get (int option);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
<i>option</i>	FORTRAN: INTEGER*4 C: int	Specifies what kind of settings is requested. Named constants define possible values of <i>option</i> : <ul style="list-style-type: none">• MKL_CBWR_BRANCH - indicates the setting of the CNR branch.• MKL_CBWR_ALL - indicates all CNR-related settings.

NOTE

This release supports no CNR settings other than branch.

Description

The `mkl_cbwr_get` function returns the requested CNR settings. If the function completes successfully and the CNR mode is turned on, `mkl_cbwr_get` returns the *specific* CNR branch.

NOTE

To turn the CNR mode on, use the `mkl_cbwr_set` function or environment variables. For more details, see the *Intel MKL User's Guide*.

Return Values

Name	Type	Description
<i>setting</i>	FORTRAN: INTEGER*4 C: int	Requested CNR settings. See Named Constants for CNR Control for a list of named constants that specify the settings. If the value of the <i>option</i> parameter is not permitted, contains the MKL_CBWR_ERR_INVALID_INPUT error code.

See Also

[Usage Examples for CNR Support Functions](#)
`mkl_cbwr_set`

[`mkl_cbwr_get_auto_branch`](#)

Automatically detects the CNR code branch for your platform.

Syntax

Fortran:

```
setting = mkl_cbwr_get_auto_branch( )
```

C:

```
int mkl_cbwr_get_auto_branch (void);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Description

The `mkl_cbwr_get_auto_branch` function uses a run-time CPU check to return a CNR branch that is optimized for the processor where the program is currently running.

Optimization Notice	
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.	
Notice revision #20110804	

Return Values

Name	Type	Description
<code>setting</code>	<code>FORTRAN: INTEGER*4</code> <code>C: int</code>	Automatically detected CNR branch. May be any <i>specific</i> branch listed in Named Constants for CNR Control .

See Also

[Usage Examples for CNR Support Functions](#)

[Using a Fortran Interface Module for Support Functions](#)

Named Constants for CNR Control

Intel MKL provides reproducible results for a certain code branch, determined by the instruction set architecture (ISA). To define CNR code branches, use the following named constants as input/output for conditional numerical reproducibility support functions. Pass named constants to the functions instead of their values.

Named Constant	Value	Description
<code>MKL_CBWR_AUTO</code>	2	CNR mode uses the standard ISA-based dispatching model while ensuring fixed cache sizes, deterministic reductions, and static scheduling CNR mode uses the branch for the following ISA:

Named Constant	Value	Description
MKL_CBWR_COMPATIBLE	3	Intel® Streaming SIMD Extensions 2 (Intel® SSE2) without rcpss/rsqrtps instructions
MKL_CBWR_SSE2	4	Intel SSE2
MKL_CBWR_SSE3	5	DEPRECATED. Intel® Streaming SIMD Extensions 3 (Intel® SSE3). This setting is kept for backward compatibility and is equivalent to MKL_CBWR_SSE2.
MKL_CBWR_SSSE3	6	Supplemental Streaming SIMD Extensions 3 (SSSE3)
MKL_CBWR_SSE4_1	7	Intel® Streaming SIMD Extensions 4-1 (SSE4-1)
MKL_CBWR_SSE4_2	8	Intel® Streaming SIMD Extensions 4-2 (SSE4-2)
MKL_CBWR_AVX	9	Intel® Advanced Vector Extensions (Intel® AVX)
MKL_CBWR_AVX2	10	Intel® Advanced Vector Extensions 2 (Intel® AVX2)

When specifying the CNR branch with the named constants, be aware of the following:

- Reproducible results are provided under [Reproducibility Conditions](#).
- Settings other than `MKL_CBWR_AUTO` or `MKL_CBWR_COMPATIBLE` are available only for Intel processors.
- Intel and Intel compatible CPUs have a few instructions, such as approximation instructions `rcpps/rsqrtps`, that may return different results. Setting the branch to `MKL_CBWR_COMPATIBLE` ensures that Intel MKL does not use these instructions and forces a single Intel SSE2 only code path to be executed.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

See Also

[Usage Examples for CNR Support Functions](#)

Reproducibility Conditions

To get reproducible results from run to run, ensure that the number of threads is fixed and constant. Specifically:

- If you are running your program on different processors, explicitly specify the number of threads.
- To ensure that your application has deterministic behavior with OpenMP* parallelization and does not adjust the number of threads dynamically at run time, set `MKL_DYNAMIC` and `OMP_DYNAMIC` to FALSE. This is especially needed if you are running your program on different systems.
- As usual, you should align your data, even in CNR mode, to obtain the best possible performance. While CNR mode also fully supports unaligned input and output data, the use of it might reduce the performance of some Intel MKL functions on earlier Intel processors. To ensure proper alignment of arrays, allocate memory for them using `mkl_malloc/mkl_calloc`.

- Conditional Numerical Reproducibility does not ensure that bitwise-identical NaN values are generated when the input data contains NaN values.
- If dynamic memory allocation fails on one run but succeeds on another run, you may fail to get reproducible results between these two runs.

See Also

[mkl_malloc](#)

[mkl_calloc](#)

Usage Examples for CNR Support Functions

The following examples illustrate usage of support functions for conditional numerical reproducibility.

Setting Automatically Detected CNR Branch in C

```
#include <mkl.h>
int main(void) {
    int my_cbwr_branch;
    /* Find the available MKL_CBWR_BRANCH automatically */
    my_cbwr_branch = mkl_cbwr_get_auto_branch();
    /* User code without Intel MKL calls */
    /* Piece of the code where CNR of Intel MKL is needed */
    /* The performance of Intel MKL functions might be reduced for CNR mode */
    if (mkl_cbwr_set(my_cbwr_branch) != MKL_CBWR_SUCCESS) {
        printf("Error in setting MKL_CBWR_BRANCH! Aborting...\n");
        return;
    }
    /* CNR calls to Intel MKL + any other code */
}
```

Setting Automatically Detected CNR Branch in Fortran

```
PROGRAM MAIN
INCLUDE 'mkl.fi'
INTEGER*4 MY_CBWR_BRANCH
C Find the available MKL_CBWR_BRANCH automatically
  MY_CBWR_BRANCH = MKL_CBWR_GET_AUTO_BRANCH()
C User code without Intel MKL calls
C Piece of the code where CNR of Intel MKL is needed
C The performance of Intel MKL functions might be reduced for CNR mode
  IF (MKL_CBWR_SET (MY_CBWR_BRANCH) .NE. MKL_CBWR_SUCCESS) THEN
    PRINT *, 'Error in setting MKL_CBWR_BRANCH! Aborting...'
    RETURN
  ENDIF
C CNR calls to Intel MKL + any other code
END
```

Use of the `mkl_cbwr_get` Function in C

```
#include <mkl.h>
int main(void) {
    int my_cbwr_branch;
    /* Piece of the code where CNR of Intel MKL is analyzed */
    my_cbwr_branch = mkl_cbwr_get(MKL_CBWR_BRANCH);
    switch (my_cbwr_branch) {
        case MKL_CBWR_AUTO:
            /* actions in case of automatic mode */
            break;
        case MKL_CBWR_SSSE3:
```

```

        /* actions for SSSE3 code */
        break;
    default:
        /* all other cases */
    }
/* User code */
}

```

Use of the mkl_cbwr_get Function in Fortran

```

PROGRAM MAIN
INCLUDE 'mkl.fi'
INTEGER*4 MY_CBWR_BRANCH
C Piece of the code where CNR of Intel MKL is analyzed
    MY_CBWR_BRANCH = MKL_CBWR_GET(MKL_CBWR_BRANCH)
    IF (MY_CBWR_BRANCH .EQ. MKL_CBWR_AUTO) THEN
C actions in case of automatic mode
    ELSE IF (MY_CBWR_BRANCH .EQ. MKL_CBWR_SSSE3) THEN
C actions for SSSE3 code
    ELSE
C all other cases
    ENDIF
C User code
    END

```

Miscellaneous

mkl_progress

Provides progress information.

Syntax

Fortran:

```
stopflag = mkl_progress( thread, step, stage )
```

C:

```
int mkl_progress (int* thread, int* step, char* stage, int lstage);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
thread	FORTRAN: INTEGER*4 C: const int*	FORTRAN: The number of the thread the progress routine is called from. 0 is passed for sequential code.

Name	Type	Description
		C: Pointer to the number of the thread the progress routine is called from. 0 is passed for sequential code.
<i>step</i>	FORTRAN: INTEGER*4 C: const int*	FORTRAN: The linear progress indicator that shows the amount of work done. Increases from 0 to the linear size of the problem during the computation. C: Pointer to the linear progress indicator that shows the amount of work done. Increases from 0 to the linear size of the problem during the computation.
<i>stage</i>	FORTRAN: CHARACTER* (*) C: const char*	Message indicating the name of the routine or the name of the computation stage the progress routine is called from.
C: <i>lstage</i> int		The length of a stage string excluding the trailing NULL character.

Description

The `mkl_progress` function is intended to track progress of a lengthy computation and/or interrupt the computation. By default this routine does nothing but the user application can redefine it to obtain the computation progress information. You can set it to perform certain operations during the routine computation, for instance, to print a progress indicator. A non-zero return value may be supplied by the redefined function to break the computation.

The progress function `mkl_progress` is regularly called from some LAPACK and DSS/PARDISO functions during the computation. Refer to a specific LAPACK or DSS/PARDISO function description to see whether the function supports this feature or not.

Return Values

Name	Type	Description
<i>stopflag</i>	FORTRAN: INTEGER C: int	The stopping flag. A non-zero flag forces the routine to be interrupted. The zero flag is the default return value.

Application Notes

Note that `mkl_progress` is a Fortran routine, that is, to redefine the progress routine from C, the name should be spelt differently, parameters should be passed by reference, and an extra parameter meaning the length of the stage string should be considered. The stage string is not terminated with the NULL character. The C interface of the progress routine is as follows:

```
int mkl_progress_( int* thread, int* step, char* stage, int lstage ); // Linux* OS or OS X*
int MKL_PROGRESS( int* thread, int* step, char* stage, int lstage ); // Windows* OS
```

Example

The following example in Fortran and C languages prints the progress information to the standard output device:

Fortran:

```
integer function mkl_progress( thread, step, stage )
integer*4 thread, step
character*(*) stage
print*, 'Thread:', thread, ',stage:', stage, ',step:', step
```

```
mkl_progress = 0
return
end
```

C:

```
#include <stdio.h>
#include <string.h>
#define BUFLEN 16
int mkl_progress_( int* ithr, int* step, char* stage, int lstage )
{
    char buf[BUFLEN];
    if( lstage >= BUFLEN ) lstage = BUFLEN-1;
    strncpy( buf, stage, lstage );
    buf[lstage] = '\0';
    printf( "In thread %i, at stage %s, steps passed %i\n", *ithr, buf, *step );
    return 0;
}
```

mkl_enable_instructions

Enables dispatching for new Intel® architectures.

Syntax

Fortran:

```
irc = mkl_enable_instructions(isa)
```

C:

```
int mkl_enable_instructions (int isa);
```

Fortran Include Files/Modules

- Include file: mkl.fi
- Module (compiled): mkl_service.mod
- Module (source): mkl_service.f90

C Include Files

- mkl.h

Input Parameters

Name	Type	Description
isa	FORTRAN: INTEGER*4 C: int	<p>The Intel instruction-set architecture (ISA) for which to enable dispatching.</p> <p>Possible values:</p> <ul style="list-style-type: none"> • MKL_ENABLE_AVX512 - Requests to enable dispatching Intel® Advanced Vector Extensions 512 (Intel® AVX-512) on Intel® Xeon® processors. • MKL_ENABLE_AVX512_MIC - Requests to enable dispatching Intel AVX-512 on Intel® Xeon Phi™ processors and coprocessors.

For more information about the Intel AVX-512 instructions, refer to <https://software.intel.com/en-us/blogs/2013/avx-512-instructions>.

Description

If automatic dispatching is not yet enabled for the specified ISA, `mkl_enable_instructions` enables Intel MKL to dispatch the instruction set to Intel processors when the library runs on hardware (or simulation) that supports this instruction set.

Return Values

Name	Type	Description
<code>irc</code>	<code>FORTRAN: INTEGER*4</code> <code>C: int</code>	<p>Value reflecting usage status of the specified instruction set:</p> <p>1 - Intel MKL uses the specified instruction set if the hardware supports it.</p> <p>0 - The request is rejected. Most likely, <code>mkl_enable_instructions</code> has been called after another Intel MKL function.</p>

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

`mkl_set_env_mode`

Sets up the mode that ignores environment settings specific to Intel MKL.

Syntax

Fortran:

```
current_mode = mkl_set_env_mode( mode )
```

C:

```
int mkl_set_env_mode(int mode);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>mode</i>	FORTRAN: INTEGER*4 C: int	Specifies what mode to set. For details, see Description . Possible values: <ul style="list-style-type: none"> • 0 - Do nothing. • Use this value to query the current environment mode. • 1 - Make Intel MKL ignore environment settings specific to the library.

Description

In the default *environment mode*, Intel MKL can control its behavior using environment variables for threading, memory management, Conditional Numerical Reproducibility, automatic offload, and so on. The `mkl_set_env_mode` function sets up the environment mode that ignores all settings specified by Intel MKL environment variables except `MIC_LD_LIBRARY_PATH` and `MKLROOT`.

Return Values

Name	Type	Description
<i>current_mode</i>	FORTRAN: INTEGER*4 C: int	Environment mode that was used before the function call: <ul style="list-style-type: none"> • 0 - Default • 1 - Ignore environment settings specific to Intel MKL.

`mkl_verbose`

Enables or disables Intel MKL Verbose mode.

Syntax

Fortran:

```
status = mkl_verbose(enable)
```

C:

```
int mkl_verbose (int enable);
```

Fortran Include Files/Modules

- Include file: `mkl.fi`
- Module (compiled): `mkl_service.mod`
- Module (source): `mkl_service.f90`

C Include Files

- `mkl.h`

Input Parameters

Name	Type	Description
<i>enable</i>	FORTRAN: INTEGER*4 C: int	Desired state of the Intel MKL Verbose mode. Indicates whether printing Intel MKL function call information should be turned on or off. Possible values:

Name	Type	Description
		<ul style="list-style-type: none">• 0 - disable the Verbose mode.• 1 - enable the Verbose mode.

Description

This function enables or disables the Intel MKL Verbose mode, in which computational functions print call description information. For details of the Verbose mode, see the *Intel MKL User's Guide*, available in the Intel® Software Documentation Library.

NOTE

The setting for the Verbose mode specified by the `mkl_verbose` function takes precedence over the setting specified by the `MKL_VERBOSE` environment variable.

Return Values

Name	Type	Description
<code>status</code>	<code>FORTRAN: INTEGER*4</code> <code>C: int</code>	<ul style="list-style-type: none">• If the requested operation completed successfully, contains previous state of the verbose mode:<ul style="list-style-type: none">• 0 - disabled• 1 - enabled• If the function failed to complete the operation because of an incorrect input parameter, equals -1.

See Also

[Intel Software Documentation Library](#)

BLACS Routines

This chapter describes the Intel® Math Kernel Library implementation of FORTRAN 77 routines from the BLACS (Basic Linear Algebra Communication Subprograms) package. These routines are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The BLACS routines make linear algebra applications both easier to program and more portable. For this purpose, they are used in Intel MKL intended for the Linux* and Windows* OSs as the communication layer of ScalAPACK and Cluster FFT.

On computers, a linear algebra matrix is represented by a two dimensional array (2D array), and therefore the BLACS operate on 2D arrays. See description of the basic [matrix shapes](#) in a special section.

The BLACS routines implemented in Intel MKL are of four categories:

- Combines
- Point to Point Communication
- Broadcast
- Support.

The [Combines](#) take data distributed over processes and combine the data to produce a result. The [Point to Point](#) routines are intended for point-to-point communication and [Broadcast](#) routines send data possessed by one process to all processes within a scope.

The [Support routines](#) perform distinct tasks that can be used for initialization, destruction, information, and miscellaneous tasks.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Matrix Shapes

The BLACS routines recognize the two most common classes of matrices for dense linear algebra. The first of these classes consists of general rectangular matrices, which in machine storage are 2D arrays consisting of m rows and n columns, with a leading dimension, lda , that determines the distance between successive columns in memory.

The *general rectangular* matrices take the following parameters as input when determining what array to operate on:

- | | |
|-----|--|
| m | (input) INTEGER. The number of matrix rows to be operated on. |
| n | (input) INTEGER. The number of matrix columns to be operated on. |
| a | (input/output) TYPE (depends on routine), array of dimension (lda, n).
A pointer to the beginning of the (sub)array to be sent. |

lda

(input) INTEGER. The distance between two elements in matrix row.

The second class of matrices recognized by the BLACS are *trapezoidal* matrices (triangular matrices are a sub-class of trapezoidal). Trapezoidal arrays are defined by *m*, *n*, and *lda*, as above, but they have two additional parameters as well. These parameters are:

uplo

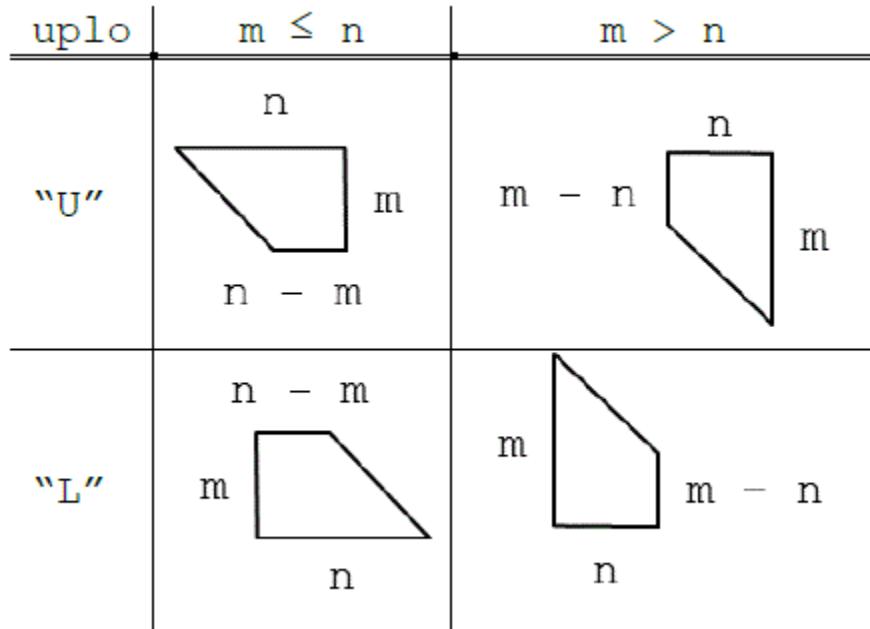
(input) CHARACTER*1 . Indicates whether the matrix is upper or lower trapezoidal, as discussed below.

diag

(input) CHARACTER*1 . Indicates whether the diagonal of the matrix is unit diagonal (will not be operated on) or otherwise (will be operated on).

The shape of the trapezoidal arrays is determined by these parameters as follows:

Trapezoidal Arrays Shapes



The packing of arrays, if required, so that they may be sent efficiently is hidden, allowing the user to concentrate on the logical matrix, rather than on how the data is organized in the system memory.

Repeatability and Coherence

Floating point computations are not exact on almost all modern architectures. This lack of precision is particularly problematic in parallel operations. Since floating point computations are inexact, algorithms are classified according to whether they are *repeatable* and to what degree they guarantee *coherence*.

- Repeatable: a routine is repeatable if it is guaranteed to give the same answer if called multiple times with the same parallel configuration and input.
- Coherent: a routine is coherent if all processes selected to receive the answer get identical results.

NOTE

Repeatability and coherence do not effect correctness. A routine may be both incoherent and non-repeatable, and still give correct output. But inaccuracies in floating point calculations may cause the routine to return differing values, all of which are equally valid.

Repeatability

Because the precision of floating point arithmetic is limited, it is not truly associative: $(a + b) + c$ might not be the same as $a + (b + c)$. The lack of exact arithmetic can cause problems whenever the possibility for reordering of floating point calculations exists. This problem becomes prevalent in parallel computing due to race conditions in message passing. For example, consider a routine which sums numbers stored on different processes. Assume this routine runs on four processes, with the numbers to be added being the process numbers themselves. Therefore, process 0 has the value 0:0, process 1 has the value 1:0, and so on.

One algorithm for the computation of this result is to have all processes send their process numbers to process 0; process 0 adds them up, and sends the result back to all processes. So, process 0 would add a number to 0:0 in the first step. If receiving the process numbers is ordered so that process 0 always receives the message from process 1 first, then 2, and finally 3, this results in a repeatable algorithm, which evaluates the expression $((0:0+1:0)+2:0)+3:0$.

However, to get the best parallel performance, it is better not to require a particular ordering, and just have process 0 add the first available number to its value and continue to do so until all numbers have been added in. Using this method, a race condition occurs, because the order of the operation is determined by the order in which process 0 receives the messages, which can be effected by any number of things. This implementation is not repeatable, because the answer can vary between invocations, even if the input is the same. For instance, one run might produce the sequence $((0:0+1:0)+2:0)+3:0$, while a subsequent run could produce $((0:0+2:0)+1:0)+3:0$. Both of these results are correct summations of the given numbers, but because of floating point roundoff, they might be different.

Coherence

A routine produces coherent output if all processes are guaranteed to produce the exact same results. Obviously, almost no algorithm involving communication is coherent if communication can change the values being communicated. Therefore, if the parallel system being studied cannot guarantee that communication between processes preserves values, no routine is guaranteed to produce coherent results.

If communication is assumed to be coherent, there are still various levels of coherent algorithms. Some algorithms guarantee coherence only if floating point operations are done in the exact same order on every node. This is *homogeneous coherence*: the result will be coherent if the parallel machine is homogeneous in its handling of floating point operations.

A stronger assertion of coherence is *heterogeneous coherence*, which does not require all processes to have the same handling of floating point operations.

In general, a routine that is homogeneous coherent performs computations redundantly on all nodes, so that all processes get the same answer only if all processes perform arithmetic in the exact same way, whereas a routine which is heterogeneous coherent is usually constrained to having one process calculate the final result, and broadcast it to all other processes.

Example of Incoherence

An incoherent algorithm is one which does not guarantee that all processes get the same result even on a homogeneous system with coherent communication. The previous example of summing the process numbers demonstrates this kind of behavior. One way to perform such a sum is to have every process broadcast its number to all other processes. Each process then adds these numbers, starting with its own. The calculations performed by each process receives would then be:

- Process 0 : $((0:0+1:0)+2:0)+3:0$
- Process 1 : $((1:0+2:0)+3:0)+0:0$
- Process 2 : $((2:0+3:0)+0:0)+1:0$
- Process 3 : $((3:0+0:0)+1:0)+0:0$

All of these results are equally valid, and since all the results might be different from each other, this algorithm is incoherent. Notice, however, that this algorithm is repeatable: each process will get the same result if the algorithm is called again on the same data.

Example of Homogeneous Coherence

Another way to perform this summation is for all processes to send their data to all other processes, and to ensure the result is not incoherent, enforce the ordering so that the calculation each node performs is $((0:0 + 1:0) + 2:0) + 3:0$. This answer is the same for all processes only if all processes do the floating point arithmetic in the same way. Otherwise, each process may make different floating point errors during the addition, leading to incoherence of the output. Notice that since there is a specific ordering to the addition, this algorithm is repeatable.

Example of Heterogeneous Coherence

In the final example, all processes send the result to process 0, which adds the numbers and broadcasts the result to the rest of the processes. Since one process does all the computation, it can perform the operations in any order and it will give coherent results as long as communication is itself coherent. If a particular order is not forced on the the addition, the algorithm will not be repeatable. If a particular order is forced, it will be repeatable.

Summary

Repeatability and coherence are separate issues which may occur in parallel computations. These concepts may be summarized as:

- Repeatability: The routine will yield the exact same result if it run multiple times on an identical problem. Each process may get a different result than the others (i.e., repeatability does not imply coherence), but that value will not change if the routine is invoked multiple times.
- Homogeneous coherence: All processes selected to possess the result will receive the exact same answer if:
 - Communication does not change the value of the communicated data.
 - All processes perform floating point arithmetic exactly the same.
- Heterogeneous coherence: All processes will receive the exact same answer if communication does not change the value of the communicated data.

In general, lack of the associative property for floating point calculations may cause both incoherence and non-repeatability. Algorithms that rely on redundant computations are at best homogeneous coherent, and algorithms in which one process broadcasts the result are heterogeneous coherent. Repeatability does not imply coherence, nor does coherence imply repeatability.

Since these issues do not effect the correctness of the answer, they can usually be ignored. However, in very specific situations, these issues may become very important. A stopping criteria should not be based on incoherent results, for instance. Also, a user creating and debugging a parallel program may wish to enforce repeatability so the exact same program sequence occurs on every run.

In the BLACS, coherence and repeatability apply only in the context of the combine operations. As mentioned above, it is possible to have communication which is incoherent (for instance, two machines which store floating point numbers differently may easily produce incoherent communication, since a number stored on machine A may not have a representation on machine B). However, the BLACS cannot control this issue. Communication is assumed to be coherent, which for communication implies that it is also repeatable.

For combine operations, the BLACS allow you to set flags indicating that you would like combines to be repeatable and/or heterogeneous coherent (see `blacs_get` and `blacs_set` for details on setting these flags).

If the BLACS are instructed to guarantee heterogeneous coherency, the BLACS restrict the topologies which can be used so that one process calculates the final result of the combine, and if necessary, broadcasts the answer to all other processes.

If the BLACS are instructed to guarantee repeatability, orderings will be enforced in the topologies which are selected. This may result in loss of performance which can range from negligible to serious depending on the application.

A couple of additional notes are in order. Incoherence and nonrepeatability can arise as a result of floating point errors, as discussed previously. This might lead you to suspect that integer calculations are always repeatable and coherent, since they involve exact arithmetic. This is true if overflow is ignored. With overflow taken into consideration, even integer calculations can display incoherence and non-repeatability. Therefore, if the repeatability or coherence flags are set, the BLACS treats integer combines the same as floating point combines in enforcing repeatability and coherence guards.

By their nature, maximization and minimization should always be repeatable. In the complex precisions, however, the real and imaginary parts must be combined in order to obtain a magnitude value used to do the comparison (this is typically $|r| + |i|$ or $\text{sqr}(r^2 + i^2)$). This allows for the possibility of heterogeneous incoherence. The BLACS therefore restrict which topologies are used for maximization and minimization in the complex routines when the heterogeneous coherence flag is set.

BLACS Combine Operations

This section describes BLACS routines that combine the data to produce a result.

In a combine operation, each participating process contributes data that is combined with other processes' data to produce a result. This result can be given to a particular process (called the *destination* process), or to all participating processes. If the result is given to only one process, the operation is referred to as a *leave-on-one* combine, and if the result is given to all participating processes the operation is referenced as a *leave-on-all* combine.

At present, three kinds of combines are supported. They are:

- element-wise summation
- element-wise absolute value maximization
- element-wise absolute value minimization

of general rectangular arrays.

Note that a combine operation combines data between processes. By definition, a combine performed across a scope of only one process does not change the input data. This is why the operations (`max/min/sum`) are specified as *element-wise*. Element-wise indicates that each element of the input array will be combined with the corresponding element from all other processes' arrays to produce the result. Thus, a 4×2 array of inputs produces a 4×2 answer array.

When the `max/min` comparison is being performed, absolute value is used. For example, -5 and 5 are equivalent. However, the returned value is unchanged; that is, it is not the absolute value, but is a signed value instead. Therefore, if you performed a BLACS absolute value maximum combine on the numbers -5, 3, 1, 8 the result would be -8.

The initial symbol ? in the routine names below masks the data type:

i	integer
s	single precision real
d	double precision real
c	single precision complex
z	double precision complex.

BLACS Combines

Routine name	Results of operation
<code>gamx2d</code>	Entries of result matrix will have the value of the greatest absolute value found in that position.
<code>gamm2d</code>	Entries of result matrix will have the value of the smallest absolute value found in that position.

Routine name	Results of operation
gsum2d	Entries of result matrix will have the summation of that position.

?gamx2d

Performs element-wise absolute value maximization.

Syntax

```
call igamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call sgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call dgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call cgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call zgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
```

Input Parameters

icontxt	INTEGER. Integer handle that indicates the context.
scope	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
top	CHARACTER*1. Communication pattern to use during the combine operation.
m	INTEGER. The number of matrix rows to be combined.
n	INTEGER. The number of matrix columns to be combined.
a	TYPE array (lda, n). Matrix to be compared with to produce the maximum.
lda	INTEGER. The leading dimension of the matrix A, that is, the distance between two successive elements in a matrix row.
rcflag	INTEGER. If <i>rcflag</i> = -1, the arrays <i>ra</i> and <i>ca</i> are not referenced and need not exist. Otherwise, <i>rcflag</i> indicates the leading dimension of these arrays, and so must be $\geq m$.
rdest	INTEGER. The process row coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.
cdest	INTEGER. The process column coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.

Output Parameters

a TYPE array (*lda*, *n*). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.

ra INTEGER array (*rcflag*, *n*).
If *rcflag* = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least *rcflag* × *n*) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

ca INTEGER array (*rcflag*, *n*).
If *rcflag* = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least *rcflag* × *n*) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

Description

This routine performs element-wise absolute value maximization, that is, each element of matrix *A* is compared with the corresponding element of the other process's matrices. Note that the value of *A* is returned, but the absolute value is used to determine the maximum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

[Examples of BLACS Routines Usage](#)

?gamn2d

Performs element-wise absolute value minimization.

Syntax

```
call igamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call sgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call dgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call cgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call zgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
```

Input Parameters

icontxt INTEGER. Integer handle that indicates the context.

scope CHARACTER*1. Indicates what scope the combine should proceed on.
Limited to ROW, COLUMN, or ALL.

top CHARACTER*1. Communication pattern to use during the combine operation.

<i>m</i>	INTEGER. The number of matrix rows to be combined.
<i>n</i>	INTEGER. The number of matrix columns to be combined.
<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Matrix to be compared with to produce the minimum.
<i>lda</i>	INTEGER. The leading dimension of the matrix <i>A</i> , that is, the distance between two successive elements in a matrix row.
<i>rcflag</i>	INTEGER. If <i>rcflag</i> = -1, the arrays <i>ra</i> and <i>ca</i> are not referenced and need not exist. Otherwise, <i>rcflag</i> indicates the leading dimension of these arrays, and so must be $\geq m$.
<i>rdest</i>	INTEGER. The process row coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.
<i>cdest</i>	INTEGER. The process column coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.

Output Parameters

<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.
<i>ra</i>	INTEGER array (<i>rcflag</i> , <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> \times <i>n</i>) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.
<i>ca</i>	INTEGER array (<i>rcflag</i> , <i>n</i>). If <i>rcflag</i> = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least <i>rcflag</i> \times <i>n</i>) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

Description

This routine performs element-wise absolute value minimization, that is, each element of matrix *A* is compared with the corresponding element of the other process's matrices. Note that the value of *A* is returned, but the absolute value is used to determine the minimum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

[Examples of BLACS Routines Usage](#)

?gsum2d

Performs element-wise summation.

Syntax

```
call igsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call sgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call dgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call cgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call zgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
<i>top</i>	CHARACTER*1. Communication pattern to use during the combine operation.
<i>m</i>	INTEGER. The number of matrix rows to be combined.
<i>n</i>	INTEGER. The number of matrix columns to be combined.
<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Matrix to be added to produce the sum.
<i>lda</i>	INTEGER. The leading dimension of the matrix <i>A</i> , that is, the distance between two successive elements in a matrix row.
<i>rdest</i>	INTEGER. The process row coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.
<i>cdest</i>	INTEGER. The process column coordinate of the process that should receive the result. If <i>rdest</i> or <i>cdest</i> = -1, all processes within the indicated scope receive the answer.

Output Parameters

<i>a</i>	TYPE array (<i>lda</i> , <i>n</i>). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.
----------	--

Description

This routine performs element-wise summation, that is, each element of matrix A is summed with the corresponding element of the other process's matrices. Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

[Examples of BLACS Routines Usage](#)

BLACS Point To Point Communication

This section describes BLACS routines for point to point communication.

Point to point communication requires two complementary operations. The *send* operation produces a message that is then consumed by the *receive* operation. These operations have various resources associated with them. The main such resource is the buffer that holds the data to be sent or serves as the area where the incoming data is to be received. The level of *blocking* indicates what correlation the return from a send/receive operation has with the availability of these resources and with the status of message.

Non-blocking

The return from the *send* or *receive* operations does not imply that the resources may be reused, that the message has been sent/received or that the complementary operation has been called. Return means only that the send/receive has been started, and will be completed at some later date. Polling is required to determine when the operation has finished.

In non-blocking message passing, the concept of *communication/computation overlap* (abbreviated C/C overlap) is important. If a system possesses C/C overlap, independent computation can occur at the same time as communication. That means a nonblocking operation can be posted, and unrelated work can be done while the message is sent/received in parallel. If C/C overlap is not present, after returning from the routine call, computation will be interrupted at some later date when the message is actually sent or received.

Locally-blocking

Return from the *send* or *receive* operations indicates that the resources may be reused. However, since this only depends on local information, it is unknown whether the complementary operation has been called. There are no locally-blocking receives: the send must be completed before the receive buffer is available for re-use.

If a receive has not been posted at the time a locally-blocking send is issued, buffering will be required to avoid losing the message. Buffering can be done on the sending process, the receiving process, or not done at all, losing the message.

Globally-blocking

Return from a globally-blocking procedure indicates that the operation resources may be reused, and that complement of the operation has at least been posted. Since the receive has been posted, there is no buffering required for globally-blocking sends: the message is always sent directly into the user's receive buffer.

Almost all processors support non-blocking communication, as well as some other level of blocking sends. What level of blocking the send possesses varies between platforms. For instance, the Intel® processors support locally-blocking sends, with buffering done on the receiving process. This is a very important distinction, because codes written assuming locally-blocking sends will hang on platforms with globally-blocking sends. Below is a simple example of how this can occur:

```
IAM = MY_PROCESS_ID()
IF (IAM .EQ. 0) THEN
    SEND TO PROCESS 1
    RECV FROM PROCESS 1
ELSE IF (IAM .EQ. 1) THEN
```

```

SEND TO PROCESS 0
RECV FROM PROCESS 0
END IF

```

If the send is globally-blocking, process 0 enters the send, and waits for process 1 to start its receive before continuing. In the meantime, process 1 starts to send to 0, and waits for 0 to receive before continuing. Both processes are now waiting on each other, and the program will never continue.

The solution for this case is obvious. One of the processes simply reverses the order of its communication calls and the hang is avoided. However, when the communication is not just between two processes, but rather involves a hierarchy of processes, determining how to avoid this kind of difficulty can become problematic.

For this reason, it was decided the BLACS would support locally-blocking sends. On systems natively supporting globally-blocking sends, non-blocking sends coupled with buffering is used to simulate locally-blocking sends. The BLACS support globally-blocking receives.

In addition, the BLACS specify that point to point messages between two given processes will be strictly ordered. If process 0 sends three messages (label them *A*, *B*, and *C*) to process 1, process 1 must receive *A* before it can receive *B*, and message *C* can be received only after both *A* and *B*. The main reason for this restriction is that it allows for the computation of message identifiers.

Note, however, that messages from different processes are not ordered. If processes 0, . . . , 3 send messages *A*, . . . , *D* to process 4, process 4 may receive these messages in any order that is convenient.

Convention

The convention used in the communication routine names follows the template ?xxyy2d, where the letter in the ? position indicates the data type being sent, xx is replaced to indicate the shape of the matrix, and the yy positions are used to indicate the type of communication to perform:

i	integer
s	single precision real
d	double precision real
c	single precision complex
z	double precision complex
ge	The data to be communicated is stored in a general rectangular matrix.
tr	The data to be communicated is stored in a trapezoidal matrix.
sd	Send. One process sends to another.
rv	Receive. One process receives from another.

BLACS Point To Point Communication

Routine name	Operation performed
gesd2d	Take the indicated matrix and send it to the destination process.
trsd2d	
gerv2d	Receive a message from the process into the matrix.
trrv2d	

As a simple example, the pseudo code given above is rewritten below in terms of the BLACS. It is further specified that the data being exchanged is the double precision vector X , which is 5 elements long.

```
CALL GRIDINFO(NPROW, NPCOL, MYPROW, MYPCOL)

IF (MYPROW.EQ.0 .AND. MYPCOL.EQ.0) THEN
    CALL DGESD2D(5, 1, X, 5, 1, 0)
    CALL DGERV2D(5, 1, X, 5, 1, 0)
ELSE IF (MYPROW.EQ.1 .AND. MYPCOL.EQ.0) THEN
    CALL DGESD2D(5, 1, X, 5, 0, 0)
    CALL DGERV2D(5, 1, X, 5, 0, 0)
END IF
```

?gesd2d

Takes a general rectangular matrix and sends it to the destination process.

Syntax

```
call igesd2d( icontxt, m, n, a, lda, rdest, cdest )
call sgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call dgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call cgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call zgesd2d( icontxt, m, n, a, lda, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rdest</i>	INTEGER.
	The process row coordinate of the process to send the message to.
<i>cdest</i>	INTEGER.
	The process column coordinate of the process to send the message to.

Description

This routine takes the indicated general rectangular matrix and sends it to the destination process located at {RDEST, CDEST} in the process grid. Return from the routine indicates that the buffer (the matrix A) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

See Also

[Examples of BLACS Routines Usage](#)

?trsd2d

Takes a trapezoidal matrix and sends it to the destination process.

Syntax

```
call itrasd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call strasd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
```

```
call dtrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ctrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ztrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>uplo, diag, m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rdest</i>	INTEGER.
	The process row coordinate of the process to send the message to.
<i>cdest</i>	INTEGER.
	The process column coordinate of the process to send the message to.

Description

This routine takes the indicated trapezoidal matrix and sends it to the destination process located at {RDEST, CDEST} in the process grid. Return from the routine indicates that the buffer (the matrix A) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

?gerv2d

Receives a message from the process into the general rectangular matrix.

Syntax

```
call igerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call sgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call dgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call cgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
call zgerv2d( icontxt, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>m, n, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER.
	The process row coordinate of the source of the message.
<i>csrc</i>	INTEGER.
	The process column coordinate of the source of the message.

Output Parameters

<i>a</i>	An array of dimension (<i>lda, n</i>) to receive the incoming message into.
----------	---

Description

This routine receives a message from process {RSRC, CSRC} into the general rectangular matrix A . This routine is globally-blocking, that is, return from the routine indicates that the message has been received into A .

See Also

[Examples of BLACS Routines Usage](#)

?trrv2d

Receives a message from the process into the trapezoidal matrix.

Syntax

```
call itrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call strrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call dtrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call ctrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
call ztrrv2d( icontxt, uplo, diag, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>uplo, diag, m, n, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER.
	The process row coordinate of the source of the message.
<i>csrc</i>	INTEGER.
	The process column coordinate of the source of the message.

Output Parameters

<i>a</i>	An array of dimension (lda, n) to receive the incoming message into.
----------	--

Description

This routine receives a message from process {RSRC, CSRC} into the trapezoidal matrix A . This routine is globally-blocking, that is, return from the routine indicates that the message has been received into A .

BLACS Broadcast Routines

This section describes BLACS broadcast routines.

A broadcast sends data possessed by one process to all processes within a scope. Broadcast, much like point to point communication, has two complementary operations. The process that owns the data to be broadcast issues a *broadcast/send*. All processes within the same scope must then issue the complementary *broadcast/receive*.

The BLACS define that both broadcast/send and broadcast/receive are *globally-blocking*. Broadcasts/receives cannot be locally-blocking since they must post a receive. Note that receives cannot be locally-blocking. When a given process can leave, a broadcast/receive operation is topology dependent, so, to avoid a hang as topology is varied, the broadcast/receive must be treated as if no process can leave until all processes have called the operation.

Broadcast/sends could be defined to be *locally-blocking*. Since no information is being received, as long as locally-blocking point to point sends are used, the broadcast/send will be locally blocking. However, defining one process within a scope to be locally-blocking while all other processes are globally-blocking adds little to the programmability of the code. On the other hand, leaving the option open to have globally-blocking broadcast/sends may allow for optimization on some platforms.

The fact that broadcasts are defined as globally-blocking has several important implications. The first is that scoped operations (broadcasts or combines) must be strictly ordered, that is, all processes within a scope must agree on the order of calls to separate scoped operations. This constraint falls in line with that already in place for the computation of message IDs, and is present in point to point communication as well.

A less obvious result is that scoped operations with SCOPE = 'ALL' must be ordered with respect to any other scoped operation. This means that if there are two broadcasts to be done, one along a column, and one involving the entire process grid, all processes within the process column issuing the column broadcast must agree on which broadcast will be performed first.

The convention used in the communication routine names follows the template ?xxyy2d, where the letter in the ? position indicates the data type being sent, xx is replaced to indicate the shape of the matrix, and the yy positions are used to indicate the type of communication to perform:

i	integer
s	single precision real
d	double precision real
c	single precision complex
z	double precision complex
ge	The data to be communicated is stored in a general rectangular matrix.
tr	The data to be communicated is stored in a trapezoidal matrix.
bs	Broadcast/send. A process begins the broadcast of data within a scope.
br	Broadcast/receive A process receives and participates in the broadcast of data within a scope.

BLACS Broadcast Routines

Routine name	Operation performed
gebs2d	Start a broadcast along a scope.
trbs2d	
gebr2d	Receive and participate in a broadcast along a scope.
trbr2d	

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain

Optimization Notice

optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

?gebs2d

Starts a broadcast along a scope for a general rectangular matrix.

Syntax

```
call igebs2d( icontxt, scope, top, m, n, a, lda )
call sgebs2d( icontxt, scope, top, m, n, a, lda )
call dgebs2d( icontxt, scope, top, m, n, a, lda )
call cgebs2d( icontxt, scope, top, m, n, a, lda )
call zgebs2d( icontxt, scope, top, m, n, a, lda )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>m, n, a, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the general rectangular matrix *A*.

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

See Also

[Examples of BLACS Routines Usage](#)

?trbs2d

Starts a broadcast along a scope for a trapezoidal matrix.

Syntax

```
call itrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call strbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call dtrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call ctrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
```

```
call ztrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>uplo</i> , <i>diag</i> , <i>m</i> , <i>n</i> , <i>a</i> , <i>lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix *A*.

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

?gebr2d

Receives and participates in a broadcast along a scope for a general rectangular matrix.

Syntax

```
call igebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call sgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call dgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call cgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call zgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>m</i> , <i>n</i> , <i>lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER. The process row coordinate of the process that called broadcast/send.
<i>csrc</i>	INTEGER. The process column coordinate of the process that called broadcast/send.

Output Parameters

a An array of dimension (lda, n) to receive the incoming message into.

Description

This routine receives and participates in a broadcast along a scope. At the end of a broadcast, all processes within the scope will possess the data in the general rectangular matrix A . Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

See Also

[Examples of BLACS Routines Usage](#)

?trbr2d

Receives and participates in a broadcast along a scope for a trapezoidal matrix.

Syntax

```
call itrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call strbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call dtrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call ctrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
call ztrbr2d( icontxt, scope, top, uplo, diag, m, n, a, lda, rsrc, csrc )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
<i>top</i>	CHARACTER*1. Indicates the communication pattern to use for the broadcast.
<i>uplo, diag, m, n, lda</i>	Describe the matrix to be sent. See Matrix Shapes for details.
<i>rsrc</i>	INTEGER.
	The process row coordinate of the process that called broadcast/send.
<i>csrc</i>	INTEGER.
	The process column coordinate of the process that called broadcast/send.

Output Parameters

a An array of dimension (lda, n) to receive the incoming message into.

Description

This routine receives and participates in a broadcast along a scope. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix A . Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

BLACS Support Routines

The support routines perform distinct tasks that can be used for:

Initialization

Destruction

Information Purposes

Miscellaneous Tasks.

Initialization Routines

This section describes BLACS routines that deal with grid/context creation, and processing before the grid/context has been defined.

BLACS Initialization Routines

Routine name	Operation performed
blacs_pinfo	Returns the number of processes available for use.
blacs_setup	Allocates virtual machine and spawns processes.
blacs_get	Gets values that BLACS use for internal defaults.
blacs_set	Sets values that BLACS use for internal defaults.
blacs_gridinit	Assigns available processes into BLACS process grid.
blacs_gridmap	Maps available processes into BLACS process grid.

blacs_pinfo

Returns the number of processes available for use.

Syntax

```
call blacs_pinfo( mypnum, nprocs )
```

Output Parameters

<i>mypnum</i>	INTEGER. An integer between 0 and (<i>nprocs</i> - 1) that uniquely identifies each process.
<i>nprocs</i>	INTEGER. The number of processes available for BLACS use.

Description

This routine is used when some initial system information is required before the BLACS are set up. On all platforms except PVM, *nprocs* is the actual number of processes available for use, that is, $nrows * ncols \leq nprocs$. In PVM, the virtual machine may not have been set up before this call, and therefore no parallel machine exists. In this case, *nprocs* is returned as less than one. If a process has been spawned via the keyboard, it receives *mypnum* of 0, and all other processes get *mypnum* of -1. As a result, the user can distinguish between processes. Only after the virtual machine has been set up via a call to `BLACS_SETUP`, this routine returns the correct values for *mypnum* and *nprocs*.

See Also

Examples of BLACS Routines Usage

[blacs_setup](#)

Allocates virtual machine and spawns processes.

Syntax

```
call blacs_setup( mypnum, nprocs )
```

Input Parameters

nprocs INTEGER. On the process spawned from the keyboard rather than from pvmspawn, this parameter indicates the number of processes to create when building the virtual machine.

Output Parameters

mypnum INTEGER. An integer between 0 and (*nprocs* - 1) that uniquely identifies each process.

nprocs INTEGER. For all processes other than spawned from the keyboard, this parameter means the number of processes available for BLACS use.

Description

This routine only accomplishes meaningful work in the PVM BLACS. On all other platforms, it is functionally equivalent to [blacs_pinfo](#). The BLACS assume a static system, that is, the given number of processes does not change. PVM supplies a dynamic system, allowing processes to be added to the system on the fly.

`blacs_setup` is used to allocate the virtual machine and spawn off processes. It reads in a file called `blacs_setup.dat`, in which the first line must be the name of your executable. The second line is optional, but if it exists, it should be a PVM spawn flag. Legal values at this time are 0 (`PvmTaskDefault`), 4 (`PvmTaskDebug`), 8 (`PvmTaskTrace`), and 12 (`PvmTaskDebug + PvmTaskTrace`). The primary reason for this line is to allow the user to easily turn on and off PVM debugging. Additional lines, if any, specify what machines should be added to the current configuration before spawning *nprocs*-1 processes to the machines in a round robin fashion.

nprocs is input on the process which has no PVM parent (that is, *mypnum*=0), and both parameters are output for all processes. So, on PVM systems, the call to `blacs_pinfo` informs you that the virtual machine has not been set up, and a call to `blacs_setup` then sets up the machine and returns the real values for *mypnum* and *nprocs*.

Note that if the file `blacs_setup.dat` does not exist, the BLACS prompt the user for the executable name, and processes are spawned to the current PVM configuration.

See Also

[Examples of BLACS Routines Usage](#)

[blacs_get](#)

Gets values that BLACS use for internal defaults.

Syntax

```
call blacs_get( icontxt, what, val )
```

Input Parameters

icontxt INTEGER. On values of *what* that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.

<i>what</i>	INTEGER. Indicates what BLACS internal(s) should be returned in <i>val</i> . Present options are:
	<ul style="list-style-type: none"> • <i>what</i> = 0 : Handle indicating default system context. • <i>what</i> = 1 : The BLACS message ID range. • <i>what</i> = 2 : The BLACS debug level the library was compiled with. • <i>what</i> = 10 : Handle indicating the system context used to define the BLACS context whose handle is <i>icontxt</i>. • <i>what</i> = 11 : Number of rings multiring broadcast topology is presently using. • <i>what</i> = 12 : Number of branches general tree broadcast topology is presently using. • <i>what</i> = 13 : Number of rings multiring combine topology is presently using. • <i>what</i> = 14 : Number of branches general tree combine topology is presently using. • <i>what</i> = 15 : Whether topologies are forced to be repeatable or not. A non-zero return value indicates that topologies are being forced to be repeatable. See Repeatability and Coherence for more information about repeatability. • <i>what</i> = 16 : Whether topologies are forced to be heterogenous coherent or not. A non-zero return value indicates that topologies are being forced to be heterogenous coherent. See Repeatability and Coherence for more information about coherence.

Output Parameters

<i>val</i>	INTEGER. The value of the BLACS internal.
------------	---

Description

This routine gets the values that the BLACS are using for internal defaults. Some values are tied to a BLACS context, and some are more general. The most common use is in retrieving a default system context for input into [blacs_gridinit](#) or [blacs_gridmap](#).

Some systems, such as MPI*, supply their own version of context. For those users who mix system code with BLACS code, a BLACS context should be formed in reference to a system context. Thus, the grid creation routines take a system context as input. If you wish to have strictly portable code, you may use [blacs_get](#) to retrieve a default system context that will include all available processes. This value is not tied to a BLACS context, so the parameter *icontxt* is unused.

[blacs_get](#) returns information on three quantities that are tied to an individual BLACS context, which is passed in as *icontxt*. The information that may be retrieved is:

- The handle of the system context upon which this BLACS context was defined
- The number of rings for TOP = 'M' (multiring broadcast/combine)
- The number of branches for TOP = 'T' (general tree broadcast/general tree gather).
- Whether topologies are being forced to be repeatable or heterogenous coherent.

See Also

[Examples of BLACS Routines Usage](#)

[blacs_set](#)

Sets values that BLACS use for internal defaults.

Syntax

```
call blacs_set( icontxt, what, val )
```

Input Parameters

<i>icontxt</i>	INTEGER. For values of <i>what</i> that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.
<i>what</i>	INTEGER. Indicates what BLACS internal(s) should be set. Present values are: <ul style="list-style-type: none"> • 1 = Set the BLACS message ID range • 11 = Number of rings for multiring broadcast topology to use • 12 = Number of branches for general tree broadcast topology to use • 13 = Number of rings for multiring combine topology to use • 14 = Number of branches for general tree combine topology to use • 15 = Force topologies to be repeatable or not • 16 = Force topologies to be heterogenous coherent or not
<i>val</i>	INTEGER. Array of dimension (*). Indicates the value(s) the internals should be set to. The specific meanings depend on <i>what</i> values, as discussed in Description below.

Description

This routine sets the BLACS internal defaults depending on *what* values:

<i>what</i> = 1	Setting the BLACS message ID range. If you wish to mix the BLACS with other message-passing packages, restrict the BLACS to a certain message ID range not to be used by the non-BLACS routines. The message ID range must be set before the first call to <code>blacs_gridinit</code> or <code>blacs_gridmap</code> . Subsequent calls will have no effect. Because the message ID range is not tied to a particular context, the parameter <i>icontxt</i> is ignored, and <i>val</i> is defined as: <code>VAL (input) INTEGER array of dimension (2)</code> <code>VAL(1)</code> : The smallest message ID (also called message type or message tag) the BLACS should use. <code>VAL(2)</code> : The largest message ID (also called message type or message tag) the BLACS should use.
<i>what</i> = 11	Set number of rings for TOP = 'M' (multiring broadcast). This quantity is tied to a context, so <i>icontxt</i> is used, and <i>val</i> is defined as: <code>VAL (input) INTEGER array of dimension (1)</code> <code>VAL(1)</code> : The number of rings for multiring topology to use.
<i>what</i> = 12	Set number of branches for TOP = 'T' (general tree broadcast). This quantity is tied to a context, so <i>icontxt</i> is used, and <i>val</i> is defined as: <code>VAL (input) INTEGER array of dimension (1)</code> <code>VAL(1)</code> : The number of branches for general tree topology to use.

what = 13 Set number of rings for `TOP = 'M'` (multiring combine). This quantity is tied to a context, so `ictxt` is used, and `val` is defined as:

`VAL` (input) INTEGER array of dimension (1)

`VAL(1)` : The number of rings for multiring topology to use.

what = 14 Set number of branches for `TOP = 'T'` (general tree gather). This quantity is tied to a context, so `ictxt` is used, and `val` is defined as:

`VAL` (input) INTEGER array of dimension (1)

`VAL(1)` : The number of branches for general tree topology to use.

what = 15 Force topologies to be repeatable or not (see [Repeatability and Coherence](#) for more information about repeatability).

`VAL` (input) INTEGER array of dimension (1)

`VAL(1) = 0` (default) Topologies are not required to be repeatable.

`VAL(1) ≠ 0` All used topologies are required to be repeatable, which might degrade performance.

what = 16 Force topologies to be heterogenous coherent or not (see [Repeatability and Coherence](#) for more information about coherence).

`VAL` (input) INTEGER array of dimension (1)

`VAL(1) = 0` (default) Topologies are not required to be heterogenous coherent.

`VAL(1) ≠ 0` All used topologies are required to be heterogenous coherent, which might degrade performance.

blacs_gridinit

Assigns available processes into BLACS process grid.

Syntax

```
call blacs_gridinit( icontxt, layout, nprow, npcol )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call <code>blacs_get</code> to obtain a default system context.
<i>layout</i>	CHARACTER*1. Indicates how to map processes to BLACS grid. Options are: <ul style="list-style-type: none"> • 'R' : Use row-major natural ordering • 'C' : Use column-major natural ordering • ELSE : Use row-major natural ordering
<i>nprov</i>	INTEGER. Indicates how many process rows the process grid should contain.

<i>npcol</i>	INTEGER. Indicates how many process columns the process grid should contain.
--------------	--

Output Parameters

<i>ictxt</i>	INTEGER. Integer handle to the created BLACS context.
--------------	---

Description

All BLACS codes must call this routine, or its sister routine `blacs_gridmap`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine creates a simple `nproc × ncol` process grid. This process grid uses the first `nproc * ncol` processes, and assigns them to the grid in a row- or column-major natural ordering. If these process-to-grid mappings are unacceptable, call `blacs_gridmap`.

See Also

[Examples of BLACS Routines Usage](#)

[blacs_get](#)

[blacs_gridmap](#)

[blacs_setup](#)

blacs_gridmap

Maps available processes into BLACS process grid.

Syntax

```
call blacs_gridmap( ictxt, usermap, ldumap, nproc, npcol )
```

Input Parameters

<i>ictxt</i>	INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call <code>blacs_get</code> to obtain a default system context.
--------------	---

<i>usermap</i>	INTEGER. Array, dimension (<i>ldumap</i> , <i>npcol</i>), indicating the process-to-grid mapping.
----------------	---

<i>ldumap</i>	INTEGER. Leading dimension of the 2D array <i>usermap</i> . <i>ldumap</i> \geq <i>nproc</i> .
---------------	---

<i>nproc</i>	INTEGER. Indicates how many process rows the process grid should contain.
--------------	---

npcol INTEGER. Indicates how many process columns the process grid should contain.

Output Parameters

icontext INTEGER. Integer handle to the created BLACS context.

Description

All BLACS codes must call this routine, or its sister routine `blacs_gridinit`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are actual processors (hardware), and they are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine allows the user to map processes to the process grid in an arbitrary manner. `usermap(i,j)` holds the process number of the process to be placed in {*i*, *j*} of the process grid. On most distributed systems, this process number is a machine defined number between 0 ... *nproc*-1. For PVM, these node numbers are the PVM TIDS (Task IDs). The `blacs_gridmap` routine is intended for an experienced user. The `blacs_gridinit` routine is much simpler. `blacs_gridinit` simply performs a gridmap where the first *nproc* * *npcol* processes are mapped into the current grid in a row-major natural ordering. If you are an experienced user, `blacs_gridmap` allows you to take advantage of your system's actual layout. That is, you can map nodes that are physically connected to be neighbors in the BLACS grid, etc. The `blacs_gridmap` routine also opens the way for *multigridding*: you can separate your nodes into arbitrary grids, join them together at some later date, and then re-split them into new grids. `blacs_gridmap` also provides the ability to make arbitrary grids or subgrids (for example, a "nearest neighbor" grid), which can greatly facilitate operations among processes that do not fall on a row or column of the main process grid.

See Also

[Examples of BLACS Routines Usage](#)

[blacs_get](#)

[blacs_gridinit](#)

[blacs_setup](#)

Destruction Routines

This section describes BLACS routines that destroy grids, abort processes, and free resources.

BLACS Destruction Routines

Routine name	Operation performed
blacs_freebuff	Frees BLACS buffer.
blacs_gridexit	Frees a BLACS context.

Routine name	Operation performed
blacs_abort	Aborts all processes.
blacs_exit	Frees all BLACS contexts and releases all allocated memory.

blacs_freebuff*Frees BLACS buffer.***Syntax**

```
call blacs_freebuff( icontxt, wait )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the BLACS context.
<i>wait</i>	INTEGER. Parameter indicating whether to wait for non-blocking operations or not. If equals 0, the operations should not be waited for; free only unused buffers. Otherwise, wait in order to free all buffers.

Description

This routine releases the BLACS buffer.

The BLACS have at least one internal buffer that is used for packing messages. The number of internal buffers depends on what platform you are running the BLACS on. On systems where memory is tight, keeping this buffer or buffers may become expensive. Call `freebuff` to release the buffer. However, the next call of a communication routine that requires packing reallocates the buffer.

The *wait* parameter determines whether the BLACS should wait for any non-blocking operations to be completed or not. If *wait* = 0, the BLACS free any buffers that can be freed without waiting. If *wait* is not 0, the BLACS free all internal buffers, even if non-blocking operations must be completed first.

blacs_gridexit*Frees a BLACS context.***Syntax**

```
call blacs_gridexit( icontxt )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the BLACS context to be freed.
----------------	---

Description

This routine frees a BLACS context.

Release the resources when contexts are no longer needed. After freeing a context, the context no longer exists, and its handle may be re-used if new contexts are defined.

blacs_abort*Aborts all processes.***Syntax**

```
call blacs_abort( icontxt, errornum )
```

Input Parameters

<i>icontext</i>	INTEGER. Integer handle that indicates the BLACS context to be aborted.
<i>errornum</i>	INTEGER. User-defined integer error number.

Description

This routine aborts all the BLACS processes, not only those confined to a particular context.

Use `blacs_abort` to abort all the processes in case of a serious error. Note that both parameters are input, but the routine uses them only in printing out the error message. The context handle passed in is not required to be a valid context handle.

blacs_exit

Frees all BLACS contexts and releases all allocated memory.

Syntax

```
call blacs_exit( continue )
```

Input Parameters

<i>continue</i>	INTEGER. Flag indicating whether message passing continues after the BLACS are done. If <i>continue</i> is non-zero, the user is assumed to continue using the machine after completing the BLACS. Otherwise, no message passing is assumed after calling this routine.
-----------------	---

Description

This routine frees all BLACS contexts and releases all allocated memory.

This routine should be called when a process has finished all use of the BLACS. The *continue* parameter indicates whether the user will be using the underlying communication platform after the BLACS are finished. This information is most important for the PVM BLACS. If *continue* is set to 0, then `pvm_exit` is called; otherwise, it is not called. Setting *continue* not equal to 0 indicates that explicit PVM send/recvs will be called after the BLACS routines are used. Make sure your code calls `pvm_exit`. PVM users should either call `blacs_exit` or explicitly call `pvm_exit` to avoid PVM problems.

See Also

[Examples of BLACS Routines Usage](#)

Informational Routines

This section describes BLACS routines that return information involving the process grid.

BLACS Informational Routines

Routine name	Operation performed
<code>blacs_gridinfo</code>	Returns information on the current grid.
<code>blacs_pnum</code>	Returns the system process number of the process in the process grid.
<code>blacs_pcoord</code>	Returns the row and column coordinates in the process grid.

blacs_gridinfo

Returns information on the current grid.

Syntax

```
call blacs_gridinfo( icontxt, nprow, npcol, myrow, mycol )
```

Input Parameters

icontxt INTEGER. Integer handle that indicates the context.

Output Parameters

nprow INTEGER. Number of process rows in the current process grid.

npcol INTEGER. Number of process columns in the current process grid.

myrow INTEGER. Row coordinate of the calling process in the process grid.

mycol INTEGER. Column coordinate of the calling process in the process grid.

Description

This routine returns information on the current grid. If the context handle does not point at a valid context, all quantities are returned as -1.

See Also

[Examples of BLACS Routines Usage](#)

blacs_pnum

Returns the system process number of the process in the process grid.

Syntax

```
call blacs_pnum( icontxt, prow, pcol )
```

Input Parameters

icontxt INTEGER. Integer handle that indicates the context.

prow INTEGER. Row coordinate of the process the system process number of which is to be determined.

pcol INTEGER. Column coordinate of the process the system process number of which is to be determined.

Description

This function returns the system process number of the process at {PROW, PCOL} in the process grid.

See Also

[Examples of BLACS Routines Usage](#)

blacs_pcoord

Returns the row and column coordinates in the process grid.

Syntax

```
call blacs_pcoord( icontxt, pnum, prow, pcol )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>pnum</i>	INTEGER. Process number the coordinates of which are to be determined. This parameter stand for the process number of the underlying machine, that is, it is a <i>tid</i> for PVM.

Output Parameters

<i>prow</i>	INTEGER. Row coordinates of the <i>pnum</i> process in the BLACS grid.
<i>pcol</i>	INTEGER. Column coordinates of the <i>pnum</i> process in the BLACS grid.

Description

Given the system process number, this function returns the row and column coordinates in the BLACS process grid.

See Also

[Examples of BLACS Routines Usage](#)

Miscellaneous Routines

This section describes `blacs_barrier` routine.

BLACS Informational Routines

Routine name	Operation performed
<code>blacs_barrier</code>	Holds up execution of all processes within the indicated scope until they have all called the routine.

`blacs_barrier`

Holds up execution of all processes within the indicated scope.

Syntax

```
call blacs_barrier( icontxt, scope )
```

Input Parameters

<i>icontxt</i>	INTEGER. Integer handle that indicates the context.
<i>scope</i>	CHARACTER*1. Parameter that indicates whether a process row (<i>scope='R'</i>), column (<i>'C'</i>), or entire grid (<i>'A'</i>) will participate in the barrier.

Description

This routine holds up execution of all processes within the indicated scope until they have all called the routine.

Examples of BLACS Routines Usage

Example. BLACS Usage. Hello World

The following routine takes the available processes, forms them into a process grid, and then has each process check in with the process at {0,0} in the process grid.

```

PROGRAM HELLO
*   -- BLACS example code --
*   Written by Clint Whaley 7/26/94
*   Performs a simple check-in type hello world
*
*   ..
*   .. External Functions ..
INTEGER BLACS_PNUM
EXTERNAL BLACS_PNUM
*
*   ..
*   .. Variable Declaration ..
INTEGER CONTEXT, IAM, NPROCS, NPROW, NPCOL, MYPROW, MYPCOL
INTEGER ICALLER, I, J, HISROW, HISCOL
*
*   Determine my process number and the number of processes in
*   machine
*
CALL BLACS_PINFO(IAM, NPROCS)
*
*   If in PVM, create virtual machine if it doesn't exist
*
IF (NPROCS .LT. 1) THEN
    IF (IAM .EQ. 0) THEN
        WRITE(*, 1000)
        READ(*, 2000) NPROCS
    END IF
    CALL BLACS_SETUP(IAM, NPROCS)
END IF
*
*   Set up process grid that is as close to square as possible
*
NPROW = INT( SQRT( REAL(NPROCS) ) )
NPCOL = NPROCS / NPROW
*
*   Get default system context, and define grid
*
CALL BLACS_GET(0, 0, CONTEXT)
CALL BLACS_GRIDINIT(CONTEXT, 'Row', NPROW, NPCOL)
CALL BLACS_GRIDINFO(CONTEXT, NPROW, NPCOL, MYPROW, MYPCOL)
*
*   If I'm not in grid, go to end of program
*
IF ( (MYPROW.GE.NPROW) .OR. (MYPCOL.GE.NPCOL) ) GOTO 30
*
*   Get my process ID from my grid coordinates
*
ICALLER = BLACS_PNUM(CONTEXT, MYPROW, MYPCOL)
*
*   If I am process {0,0}, receive check-in messages from
*   all nodes
*
IF ( (MYPROW.EQ.0) .AND. (MYPCOL.EQ.0) ) THEN

```

```

      WRITE(*,*) ' '
      DO 20 I = 0, NPROW-1
          DO 10 J = 0, NPCOL-1

              IF ( (I.NE.0) .OR. (J.NE.0) ) THEN
                  CALL IGERV2D(CONTXT, 1, 1, ICALLER, 1, I, J)
              END IF
*
*           Make sure ICALLER is where we think in process grid
*
              CALL BLACS_PCOORD(CONTXT, ICALLER, HISROW, HISCOL)
              IF ( (HISROW.NE.I) .OR. (HISCOL.NE.J) ) THEN
                  WRITE(*,*) 'Grid error! Halting . . .'
                  STOP
              END IF
              WRITE(*, 3000) I, J, ICALLER

10         CONTINUE
20         CONTINUE
            WRITE(*,*) ' '
            WRITE(*,*) 'All processes checked in. Run finished.'
*
*           All processes but {0,0} send process ID as a check-in
*
            ELSE
                CALL IGESD2D(CONTXT, 1, 1, ICALLER, 1, 0, 0)
            END IF

30         CONTINUE

            CALL BLACS_EXIT(0)

1000    FORMAT('How many processes in machine?')
2000    FORMAT(I)
3000    FORMAT('Process ',i2,',',i2,') (node number =',I,
$           ') has checked in.')
*
            STOP
        END

```

Example. BLACS Usage. PROCMAP

This routine maps processes to a grid using blacs_gridmap.

```

SUBROUTINE PROCMAP(CONTEXT, MAPPING, BEGPROC, NPROW, NPCOL, IMAP)
*
*           -- BLACS example code --
*
*           Written by Clint Whaley 7/26/94

```

```
*      ..
*      .. Scalar Arguments ..
INTEGER CONTEXT, MAPPING, BEGPROC, NPROW, NPCOL

*
*      ..
*      .. Array Arguments ..
INTEGER IMAP(NPROW, *)
*
*      ..

*
* Purpose
* ======
* PROCMAP maps NPROW*NPCOL processes starting from process BEGPROC to
* the grid in a variety of ways depending on the parameter MAPPING.

*
* Arguments
*
* ======
*
* CONTEXT      (output) INTEGER
* This integer is used by the BLACS to indicate a context.
* A context is a universe where messages exist and do not
* interact with other context's messages. The context
* includes the definition of a grid, and each process's
* coordinates in it.
*
* MAPPING      (input) INTEGER
* Way to map processes to grid. Choices are:
* 1 : row-major natural ordering
* 2 : column-major natural ordering
*
* BEGPROC      (input) INTEGER
* The process number (between 0 and NPROCS-1) to use as
*
* {0,0}. From this process, processes will be assigned
* to the grid as indicated by MAPPING.
*
* NPROW        (input) INTEGER
* The number of process rows the created grid
*
* should have.
*
* NPCOL        (input) INTEGER
* The number of process columns the created grid
*
* should have.
*
* IMAP          (workspace) INTEGER array of dimension (NPROW, NPCOL)
* Workspace, where the array which maps the
*
* processes to the grid will be stored for the
* call to GRIDMAP.
*
* =====
*
*      ..
*      .. External Functions ..
INTEGER BLACS_PNUM

EXTERNAL BLACS_PNUM
```

```
*      ..
*      .. External Subroutines ..
EXTERNAL BLACS_PINFO, BLACS_GRIDINIT, BLACS_GRIDMAP
*
*      ..
*      .. Local Scalars ..
INTEGER TMPCONTXT, NPROCS, I, J, K

*
*      ..
*      .. Executable Statements ..
*

*      See how many processes there are in the system
*
CALL BLACS_PINFO( I, NPROCS )

IF (NPROCS-BEGPROC .LT. NPROW*NPCOL) THEN
    WRITE(*,*) 'Not enough processes for grid'
    STOP
END IF

*      Temporarily map all processes into 1 x NPROCS grid

*
CALL BLACS_GET( 0, 0, TMPCONTXT )
CALL BLACS_GRIDINIT( TMPCONTXT, 'Row', 1, NPROCS )
K = BEGPROC

*
*      If we want a row-major natural ordering

*
IF (MAPPING .EQ. 1) THEN

    DO I = 1, NPROW
        DO J = 1, NPCOL
            IMAP(I, J) = BLACS_PNUM(TMPCONTXT, 0, K)
            K = K + 1
        END DO
    END DO

*
*      If we want a column-major natural ordering

*
ELSE IF (MAPPING .EQ. 2) THEN

    DO J = 1, NPCOL
        DO I = 1, NPROW
            IMAP(I, J) = BLACS_PNUM(TMPCONTXT, 0, K)
            K = K + 1
        END DO
    END DO
ELSE

    WRITE(*,*) 'Unknown mapping.'
```

```

        STOP
        END IF

*
*     Free temporary context
*
        CALL BLACS_GRIDEXIT(TMPCONTEXT)
*
*     Apply the new mapping to form desired context
*
        CALL BLACS_GET( 0, 0, CONTEXT )
        CALL BLACS_GRIDMAP( CONTEXT, IMAP, NPROW, NPROW, NPCOL )




        RETURN
        END

```

Example. BLACS Usage. PARALLEL DOT PRODUCT

This routine does a bone-headed parallel double precision dot product of two vectors. Arguments are input on process {0,0}, and output everywhere else.

```

DOUBLE PRECISION FUNCTION PDDOT( CONTEXT, N, X, Y )
*
* -- BLACS example code --
*
* Written by Clint Whaley 7/26/94
* ..
* .. Scalar Arguments ..
INTEGER CONTEXT, N
*
* ..
* .. Array Arguments ..
DOUBLE PRECISION X(*), Y(*)
*
* ..
*
* Purpose
* ======
* PDDOT is a restricted parallel version of the BLAS routine
* DDOT. It assumes that the increment on both vectors is one,
* and that process {0,0} starts out owning the vectors and
*
* has N. It returns the dot product of the two N-length vectors
* X and Y, that is, PDDOT = X' Y.
*
* Arguments
* ======
*
* CONTEXT      (input) INTEGER
* This integer is used by the BLACS to indicate a context.
* A context is a universe where messages exist and do not
* interact with other context's messages. The context
* includes the definition of a grid, and each process's
* coordinates in it.
*
* N           (input/output) INTEGER
* The length of the vectors X and Y. Input
* for {0,0}, output for everyone else.

```

```

*
*   X          (input/output) DOUBLE PRECISION array of dimension (N)
*   The vector X of PDDOT = X' Y. Input for {0,0},
*   output for everyone else.
*
*   Y          (input/output) DOUBLE PRECISION array of dimension (N)
*   The vector Y of PDDOT = X' Y. Input for {0,0},
*   output for everyone else.
*
*=====
*
*
*   ..
*   .. External Functions ..
DOUBLE PRECISION DDOT

EXTERNAL DDOT

*
*   ..
*   .. External Subroutines ..
EXTERNAL BLACS_GRIDINFO, DGEBS2D, DGEBR2D, DGSUM2D
*
*   ..
*   .. Local Scalars ..
INTEGER IAM, NPROCS, NPROW, NPCOL, MYPROW, MYPCOL, I, LN

DOUBLE PRECISION LDDOT

*
*   ..
*   .. Executable Statements ..

* Find out what grid has been set up, and pretend it is 1-D
*
CALL BLACS_GRIDINFO( CONXT, NPROW, NPCOL, MYPROW, MYPCOL )

IAM = MYPROW*NPCOL + MYPCOL
NPROCS = NPROW * NPCOL
*
* Temporarily map all processes into 1 x NPROCS grid
*
*
CALL BLACS_GET( 0, 0, TMPCONTXT )
CALL BLACS_GRIDINIT( TMPCONTXT, 'Row', 1, NPROCS )
K = BEGPROC

*
* Do bone-headed thing, and just send entire X and Y to
*
* everyone
*
IF ( (MYPROW.EQ.0) .AND. (MYPCOL.EQ.0) ) THEN
    CALL IGEBS2D( CONXT, 'All', 'i-ring', 1, 1, N, 1 )
    CALL DGEBS2D( CONXT, 'All', 'i-ring', N, 1, X, N )

```

```

    CALL DGEBS2D(CONTXT, 'All', 'i-ring', N, 1, Y, N )
ELSE
    CALL IGEBR2D(CONTXT, 'All', 'i-ring', 1, 1, N, 1, 0, 0 )
    CALL DGEBR2D(CONTXT, 'All', 'i-ring', N, 1, X, N, 0, 0 )
    CALL DGEBR2D(CONTXT, 'All', 'i-ring', N, 1, Y, N, 0, 0 )
ENDIF
*
*   Find out the number of local rows to multiply (LN), and
*
*   where in vectors to start (I)
*
*
LN = N / NPROCS

I = 1 + IAM * LN
*
Last process does any extra rows
*
IF (IAM .EQ. NPROCS-1) LN = LN + MOD(N, NPROCS)
*
Figure dot product of my piece of X and Y
*
LDDOT = DDOT( LN, X(I), 1, Y(I), 1 )
*
Add local dot products to get global dot product;

*
give all procs the answer
*
*
CALL DGSUM2D( CONTEXT, 'All', '1-tree', 1, 1, LDDOT, 1, -1, 0 )

PDDOT = LDDOT

RETURN

END

```

Example. BLACS Usage. PARALLEL MATRIX INFINITY NORM

This routine does a parallel infinity norm on a distributed double precision matrix. Unlike the PDDOT example, this routine assumes the matrix has already been distributed.

```

DOUBLE PRECISION FUNCTION PDINFNRM(CONTXT, LM, LN, A, LDA, WORK)
*
*   -- BLACS example code --
*
*   Written by Clint Whaley.
*
*   ..
*   .. Scalar Arguments ..
INTEGER CONTEXT, LM, LN, LDA
*
*   ..
*   .. Array Arguments ..

```

```
DOUBLE PRECISION A(LDA, *), WORK(*)
*
*
* Purpose
* ======
* Compute the infinity norm of a distributed matrix, where
* the matrix is spread across a 2D process grid. The result is
* left on all processes.

*
* Arguments

* =====
*
* CONTEXT      (input) INTEGER
* This integer is used by the BLACS to indicate a context.
* A context is a universe where messages exist and do not
* interact with other context's messages. The context
* includes the definition of a grid, and each process's
* coordinates in it.
*
* LM           (input) INTEGER
* Number of rows of the global matrix owned by this
* process.
*
* LN           (input) INTEGER
* Number of columns of the global matrix owned by this
* process.
*
* A            (input) DOUBLE PRECISION, dimension (LDA,N)
* The matrix whose norm you wish to compute.

*
* LDA          (input) INTEGER
* Leading Dimension of A.

*
* WORK         (temporary) DOUBLE PRECISION array, dimension (LM)
* Temporary work space used for summing rows.

*
* .. External Subroutines ..
EXTERNAL BLACS_GRIDINFO, DGEBS2D, DGEBR2D, DGSUM2D, DGAMX2D

*
* ..
* .. External Functions ..
INTEGER IDAMAX
DOUBLE PRECISION DASUM
*
* .. Local Scalars ..
INTEGER NPROW, NPCOL, MYROW, MYCOL, I, J

DOUBLE PRECISION MAX

*
* .. Executable Statements ..
```

```
*      Get process grid information
*
CALL BLACS_GRIDINFO( CONTXT, NPROW, NPCOL, MYPROW, MYPOL )
```

```
*      Add all local rows together
```

```
*
```

```
DO 20 I = 1, LM
```

```
      WORK(I) = DASUM(LN, A(I,1), LDA)
```

```
20  CONTINUE
```

```
*
```

```
*      Find sum of global matrix rows and store on column 0 of
```

```
*
```

```
process grid
```

```
*
```

```
CALL DGSUM2D(CONTXT, 'Row', '1-tree', LM, 1, WORK, LM, MYROW, 0)
```

```
*
```

```
*      Find maximum sum of rows for supnorm
```

```
*
```

```
IF (MYCOL .EQ. 0) THEN
```

```
    MAX = WORK(IDAMAX(LM,WORK,1))
```

```
    IF (LM .LT. 1) MAX = 0.0D0
```

```
        CALL DGAMX2D(CONTXT, 'Col', 'h', 1, 1, MAX, 1, I, I, -1, -1, 0)
```

```
END IF
```

```
*
```

```
*      Process column 0 has answer; send answer to all nodes
```

```
*
```

```
IF (MYCOL .EQ. 0) THEN
```

```
    CALL DGEBS2D(CONTXT, 'Row', ' ', 1, 1, MAX, 1)
```

```
ELSE
```

```
    CALL DGEBR2D(CONTXT, 'Row', ' ', 1, 1, MAX, 1, 0, 0)
```

```
END IF
```

```
*
```

```
PDINFNRM = MAX
```

```
*          RETURN
*
*      End of PDINFNRM
*
*      END
```


Data Fitting Functions

Data Fitting functions in Intel® MKL provide spline-based interpolation capabilities that you can use to approximate functions, function derivatives or integrals, and perform cell search operations.

The Data Fitting component is task based. The task is a data structure or descriptor that holds the parameters related to a specific Data Fitting operation. You can modify the task parameters using the task editing functionality of the library.

For definition of the implemented operations, see [Mathematical Conventions](#).

Data Fitting routines use the following workflow to process a task:

1. Create a task or multiple tasks.
2. Modify the task parameters.
3. Perform a Data Fitting computation.
4. Destroy the task or tasks.

All Data Fitting functions fall into the following categories:

Task Creation and Initialization Routines - routines that create a new Data Fitting task descriptor and initialize the most common parameters, such as partition of the interpolation interval, values of the vector-valued function, and the parameters describing their structure.

Task Configuration Routines - routines that set, modify, or query parameters in an existing Data Fitting task.

Computational Routines - routines that perform Data Fitting computations, such as construction of a spline, interpolation, computation of derivatives and integrals, and search.

Task Destructors - routines that delete Data Fitting task descriptors and deallocate resources.

You can access the Data Fitting routines through the Fortran and C89/C99 language interfaces. You can also use the C89 interface with more recent versions of C/C++, or the Fortran 90 interface with programs written in Fortran 95.

The \${MKL}/include directory of the Intel® MKL contains the following Data Fitting header files:

- C/C++: mkl_df.h
- Fortran: mkl_df.f90 and mkl_df.f77

You can find examples that demonstrate C/C++ and Fortran usage of Data Fitting routines in the \${MKL}/examples/datafittingc directory and the \${MKL}/examples/datafittingf directory, respectively.

Naming Conventions

The Fortran interfaces of the Data Fitting functions are in lowercase, while the names of the types and constants are in uppercase.

The C/C++ interface of the Data Fitting functions, types, and constants are case-sensitive and can be in lowercase, uppercase, and mixed case.

The names of all routines have the following structure:

df[datatype]<base_name>

where

- df is a prefix indicating that the routine belongs to the Data Fitting component of Intel MKL.
- [datatype] field specifies the type of the input and/or output data and can be s (for the single precision real type), d (for the double precision real type), or i (for the integer type). This field is omitted in the names of the routines that are not data type dependent.

- <base_name> field specifies the functionality the routine performs. For example, this field can be NewTask1D, Interpolate1D, or DeleteTask.

Data Types

The Data Fitting component provides routines for processing single and double precision real data types. The results of cell search operations are returned as a generic integer data type.

All Data Fitting routines use the following data type:

Type	Data Object
Fortran: TYPE(DF_TASK)	Pointer to a task
C: DFTaskPtr	

NOTE

The actual size of the generic integer type is platform-dependent. Before compiling your application, you need to set an appropriate byte size for integers. For details, see section *Using the ILP64 Interface vs. LP64 Interface* of the Intel® MKL User's Guide.

Mathematical Conventions for Data Fitting Functions

This section explains the notation used for Data Fitting function descriptions. Spline notations are based on the terminology and definitions of [deBoor2001]. The Subbotin quadratic spline definition follows the conventions of [StechSub76]. The quasi-uniform partition definition is based on [Schumaker2007].

Mathematical Notation in the Data Fitting Component

Concept	Mathematical Notation
Partition of interpolation interval $[a, b]$, where	$\{x_i\}_{i=1,\dots,n}$, where $a = x_1 < x_2 < \dots < x_n = b$
<ul style="list-style-type: none"> • x_i denotes breakpoints. • $[x_i, x_{i+1})$ denotes a sub-interval (cell) of size $\Delta_i = x_{i+1} - x_i$. 	
Quasi-uniform partition of interpolation interval $[a, b]$	<p>Partition $\{x_i\}_{i=1,\dots,n}$ which meets the constraint with a constant C defined as</p> $1 \leq M / m \leq C,$ <p>where</p> <ul style="list-style-type: none"> • $M = \max_{i=1,\dots,n-1} (\Delta_i)$ • $m = \min_{i=1,\dots,n-1} (\Delta_i)$ • $\Delta_i = x_{i+1} - x_i$
Vector-valued function of dimension p being fit	$f(x) = (f_1(x), \dots, f_p(x))$
Piecewise polynomial (PP) function f of order $k+1$	$f(x) := P_i(x), \text{ if } x \in [x_i, x_{i+1}), i = 1, \dots, n-1$ <p>where</p> <ul style="list-style-type: none"> • $\{x_i\}_{i=1,\dots,n}$ is a strictly increasing sequence of breakpoints. • $P_i(x) = c_{i,0} + c_{i,1}(x - x_i) + \dots + c_{i,k}(x - x_i)^k$ is a polynomial of degree k (order $k+1$) over the interval $x \in [x_i, x_{i+1})$.

Concept	Mathematical Notation
Function p agrees with function f at the points $\{x_i\}_{i=1,\dots,n}$.	For every point ζ in sequence $\{x_i\}_{i=1,\dots,n}$ that occurs m times, the equality $p^{(i-1)}(\zeta) = f^{(i-1)}(\zeta)$ holds for all $i = 1, \dots, m$, where $p^{(i)}(t)$ is the derivative of the i -th order.
The k -th divided difference of function f at points x_i, \dots, x_{i+k} . This difference is the leading coefficient of the polynomial of order $k+1$ that agrees with f at x_i, \dots, x_{i+k} .	$[x_i, \dots, x_{i+k}]f$ In particular, <ul style="list-style-type: none"> • $[x_1]f = f(x_1)$ • $[x_1, x_2]f = (f(x_1) - f(x_2)) / (x_1 - x_2)$
A k -order derivative of interpolant $f(x)$ at interpolation site τ .	$f^{(k)}(\tau)$

Interpolants to the Function f at x_1, \dots, x_n and Boundary Conditions

Concept	Mathematical Notation
Linear interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i),$ where <ul style="list-style-type: none"> • $x \in [x_i, x_{i+1}]$ • $c_{1,i} = f(x_i)$ • $c_{2,i} = [x_i, x_{i+1}]f$ • $i = 1, \dots, n-1$
Piecewise parabolic interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2, x \in [x_i, x_{i+1}]$ Coefficients $c_{1,i}, c_{2,i}$, and $c_{3,i}$ depend on the conditions: <ul style="list-style-type: none"> • $P_i(x_i) = f(x_i)$ • $P_i(x_{i+1}) = f(x_{i+1})$ • $P_i((x_{i+1} + x_i) / 2) = v_{i+1}$ where parameter v_{i+1} depends on the interpolant being continuously differentiable: $P_{i-1}^{(1)}(x_i) = P_i^{(1)}(x_i)$
Piecewise parabolic Subbotin interpolant	$P(x) = P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + d_{3,i}((x - t_i)_+)^2,$ where <ul style="list-style-type: none"> • $x \in [t_i, t_{i+1}]$ • $\{t_i\}_{i=1,\dots,n+1}$ is a sequence of knots such that <ul style="list-style-type: none"> • $t_1 = x_1, t_{n+1} = x_n$ • $t_i \in (x_{i-1}, x_i), i = 2, \dots, n$ • $x_+ = f(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$ Coefficients $c_{1,i}, c_{2,i}, c_{3,i}$, and $d_{3,i}$ depend on the following conditions: <ul style="list-style-type: none"> • $P_i(x_i) = f(x_i), P_i(x_{i+1}) = f(x_{i+1})$ • $P(x)$ is a continuously differentiable polynomial of the second degree on $[t_i, t_{i+1}], i = 1, \dots, n$.
Piecewise cubic Hermite interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3,$ where

Concept	Mathematical Notation
	<ul style="list-style-type: none"> • $x \in [x_i, x_{i+1})$ • $c_{1,i} = f(x_i)$ • $c_{2,i} = s_i$ • $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ • $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ • $i = 1, \dots, n-1$ • $s_i = f^{(1)}(x_i)$
Piecewise cubic Bessel interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3,$
	where
	<ul style="list-style-type: none"> • $x \in [x_i, x_{i+1})$ • $c_{1,i} = f(x_i)$ • $c_{2,i} = s_i$ • $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ • $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ • $i = 1, \dots, n-1$ • $s_i = (\Delta x_i[x_{i-1}, x_i]f + \Delta x_{i-1}[x_i, x_{i+1}]f) / (\Delta x_i + \Delta x_{i+1})$
Piecewise cubic Akima interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3,$
	where
	<ul style="list-style-type: none"> • $x \in [x_i, x_{i+1})$ • $c_{1,i} = f(x_i)$ • $c_{2,i} = s_i$ • $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ • $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ • $i = 1, \dots, n-1$ • $s_i = (w_{i+1}[x_{i-1}, x_i]f + w_{i-1}[x_i, x_{i+1}]f) / (w_{i+1} + w_{i-1}),$
	where
	$w_i = [x_i, x_{i+1}]f - [x_{i-1}, x_i]f $
Piecewise natural cubic interpolant	$P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3,$
	where
	<ul style="list-style-type: none"> • $x \in [x_i, x_{i+1})$ • $c_{1,i} = f(x_i)$ • $c_{2,i} = s_i$ • $c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$ • $c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]f) / (\Delta x_i)^2$ • $i = 1, \dots, n-1$ • Parameter s_i depends on the condition that the interpolant is twice continuously differentiable: $P_{i-1}^{(2)}(x_i) = P_i^{(2)}(x_i).$
Not-a-knot boundary condition.	Parameters s_1 and s_n provide $P_1 = P_2$ and $P_{n-1} = P_n$, so that the first and the last interior breakpoints are inactive.
Free-end boundary condition.	$f''(x_1) = f''(x_n) = 0$

Concept	Mathematical Notation
Look-up interpolator for discrete set of points $(x_1, y_1), \dots, (x_n, y_n)$.	$y(x) = \begin{cases} y_1, & \text{if } x = x_1 \\ y_2, & \text{if } x = x_2 \\ \dots \\ y_n, & \text{if } x = x_n \\ \text{error, otherwise} \end{cases}$
Step-wise constant continuous right interpolator.	$y(x) = \begin{cases} y_1, & \text{if } x_1 \leq x < x_2 \\ y_2, & \text{if } x_2 \leq x < x_3 \\ \dots \\ y_{n-1}, & \text{if } x_{n-1} \leq x < x_n \\ y_n, & \text{if } x = x_n \end{cases}$
Step-wise constant continuous left interpolator.	$y(x) = \begin{cases} y_1, & \text{if } x = x_1 \\ y_2, & \text{if } x_1 < x \leq x_2 \\ y_3, & \text{if } x_2 < x \leq x_3 \\ \dots \\ y_n, & \text{if } x_{n-1} < x \leq x_n \end{cases}$

Data Fitting Usage Model

Consider an algorithm that uses the Data Fitting functions. Typically, such algorithms consist of four steps or stages:

1. Create a task. You can call the Data Fitting function several times to create multiple tasks.

```
status = dfdNewTask1D( &task, nx, x, xhint, ny, y, yhint );
```

2. Modify the task parameters.

```
status = dfdEditPPSpline1D( task, s_order, c_type, bc_type, bc, ic_type, ic,
                           scoeff, scoeffhint );
```

3. Perform Data Fitting spline-based computations. You may reiterate steps 2-3 as needed.

```
status = dfdInterpolate1D(task, estimate, method, nsite, site, sitehint, ndorder,
                           dorder, datahint, r, rhint, cell );
```

4. Destroy the task or tasks.

```
status = dfDeleteTask( &task );
```

See Also

[Data Fitting Usage Examples](#)

Data Fitting Usage Examples

The examples below illustrate several operations that you can perform with Data Fitting routines.

You can get both C and Fortran source code in the `.\examples\datafittingc` subdirectory and the `.\examples\datafittingf` subdirectory of the Intel MKL installation directory to perform similar examples.

The following C example demonstrates the construction of a linear spline using Data Fitting routines. The spline approximates a scalar function defined on non-uniform partition. The coefficients of the spline are returned as a one-dimensional array:

C Example of Linear Spline Construction

```
#include "mkl.h"
#define N 500                                /* Size of partition, number of breakpoints */
#define SPLINE_ORDER DF_PP_LINEAR /* Linear spline to construct */

int main()
{
    int status;                      /* Status of a Data Fitting operation */
    DFTaskPtr task;                  /* Data Fitting operations are task based */

    /* Parameters describing the partition */
    MKL_INT nx;                     /* The size of partition x */
    double x[N];                    /* Partition x */
    MKL_INT xhint;                  /* Additional information about the structure of breakpoints */

    /* Parameters describing the function */
    MKL_INT ny;                     /* Function dimension */
    double y[N];                    /* Function values at the breakpoints */
    MKL_INT yhint;                  /* Additional information about the function */

    /* Parameters describing the spline */
    MKL_INT s_order;                /* Spline order */
    MKL_INT s_type;                 /* Spline type */
    MKL_INT ic_type;                /* Type of internal conditions */
    double* ic;                     /* Array of internal conditions */
    MKL_INT bc_type;                /* Type of boundary conditions */
    double* bc;                     /* Array of boundary conditions */

    double scoeff[(N-1)* SPLINE_ORDER]; /* Array of spline coefficients */
    MKL_INT scoeffhint;             /* Additional information about the coefficients */

    /* Initialize the partition */
    nx = N;
    /* Set values of partition x */
    ...
    xhint = DF_NO_HINT;            /* No additional information about the function is provided.
                                    By default, the partition is non-uniform. */
    /* Initialize the function */
    ny = 1;                       /* The function is scalar. */

    /* Set function values */
    ...
    yhint = DF_NO_HINT;            /* No additional information about the function is provided. */

    /* Create a Data Fitting task */
    status = dfdNewTask1D( &task, nx, x, xhint, ny, y, yhint );

    /* Check the Data Fitting operation status */
    ...

    /* Initialize spline parameters */
    s_order = DF_PP_LINEAR;        /* Spline is of the second order. */
    s_type = DF_PP_DEFAULT;        /* Spline is of the default type. */

    /* Define internal conditions for linear spline construction (none in this example) */
    ic_type = DF_NO_IC;
```

```

ic = NULL;

/* Define boundary conditions for linear spline construction (none in this example) */
bc_type = DF_NO_BC;
bc = NULL;
scoeffhint = DF_NO_HINT; /* No additional information about the spline. */

/* Set spline parameters in the Data Fitting task */
status = dfdEditPPSpline1D( task, s_order, s_type, bc_type, bc, ic_type,
                            ic, scoeff, scoeffhint );

/* Check the Data Fitting operation status */
...

/* Use a standard computation method to construct a linear spline: */
/* P_i(x) = c_{i,0} + c_{i,1}(x - x_i), i=0,..., N-2 */
/* The library packs spline coefficients to array scoeff. */
/* scoeff[2*i+0]=c_{i,0} and scoeff[2*i+1]=c_{i,1}, i=0,..., N-2 */
status = dfdConstruct1D( task, DF_PP_SPLINE, DF_METHOD_STD );

/* Check the Data Fitting operation status */
...

/* Process spline coefficients */
...

/* Deallocate Data Fitting task resources */
status = dfDeleteTask( &task ) ;

/* Check the Data Fitting operation status */
...
return 0 ;
}

```

The following C example demonstrates cubic spline-based interpolation using Data Fitting routines. In this example, a scalar function defined on non-uniform partition is approximated by Bessel cubic spline using not-a-knot boundary conditions. Once the spline is constructed, you can use the spline to compute spline values at the given sites. Computation results are packed by the Data Fitting routine in row-major format.

C Example of Cubic Spline-Based Interpolation

```

#include "mkl.h"

#define NX 100           /* Size of partition, number of breakpoints */
#define NSITE 1000        /* Number of interpolation sites */
#define SPLINE_ORDER DF_PP_CUBIC /* A cubic spline to construct */

int main()
{
    int status;          /* Status of a Data Fitting operation */
    DFTaskPtr task;      /* Data Fitting operations are task based */

    /* Parameters describing the partition */
    MKL_INT nx;          /* The size of partition x */
    double x[NX];         /* Partition x */
    MKL_INT xhint;        /* Additional information about the structure of breakpoints */

    /* Parameters describing the function */
    MKL_INT ny;          /* Function dimension */
    double y[NX];         /* Function values at the breakpoints */
    MKL_INT yhint;        /* Additional information about the function */

```

```

/* Parameters describing the spline */
MKL_INT s_order;      /* Spline order */
MKL_INT s_type;       /* Spline type */
MKL_INT ic_type;     /* Type of internal conditions */
double* ic;           /* Array of internal conditions */
MKL_INT bc_type;     /* Type of boundary conditions */
double* bc;           /* Array of boundary conditions */

double scoeff[(NX-1)* SPLINE_ORDER];    /* Array of spline coefficients */
MKL_INT scoeffhint;                    /* Additional information about the coefficients */

/* Parameters describing interpolation computations */
MKL_INT nsite;          /* Number of interpolation sites */
double site[NSITE];      /* Array of interpolation sites */
MKL_INT sitehint;        /* Additional information about the structure of
                           interpolation sites */

MKL_INT ndorder, dorder;    /* Parameters defining the type of interpolation */

double* datahint;         /* Additional information on partition and interpolation sites */

double r[NSITE];          /* Array of interpolation results */
MKL_INT rhint;            /* Additional information on the structure of the results */
MKL_INT* cell;             /* Array of cell indices */

/* Initialize the partition */
nx = NX;

/* Set values of partition x */
...
xhint = DF_NON_UNIFORM_PARTITION; /* The partition is non-uniform. */

/* Initialize the function */
ny = 1;                  /* The function is scalar. */

/* Set function values */
...
yhint = DF_NO_HINT;      /* No additional information about the function is provided. */

/* Create a Data Fitting task */
status = dfdNewTaskID( &task, nx, x, xhint, ny, y, yhint );

/* Check the Data Fitting operation status */
...

/* Initialize spline parameters */
s_order = DF_PP_CUBIC;    /* Spline is of the fourth order (cubic spline). */
s_type = DF_PP_BESSEL;    /* Spline is of the Bessel cubic type. */

/* Define internal conditions for cubic spline construction (none in this example) */
ic_type = DF_NO_IC;
ic = NULL;

/* Use not-a-knot boundary conditions. In this case, the first and the last
   interior breakpoints are inactive, no additional values are provided. */
bc_type = DF_BC_NOT_A_KNOT;
bc = NULL;
scoeffhint = DF_NO_HINT;    /* No additional information about the spline. */

```

```

/* Set spline parameters in the Data Fitting task */
status = dfdEditPPSpline1D( task, s_order, s_type, bc_type, bc, ic_type,
                            ic, scoeff, scoeffhint );

/* Check the Data Fitting operation status */
...

/* Use a standard method to construct a cubic Bessel spline: */
/*  $P_i(x) = c_{i,0} + c_{i,1}(x - x_i) + c_{i,2}(x - x_i)^2 + c_{i,3}(x - x_i)^3$ , */
/* The library packs spline coefficients to array scoeff: */
/* scoeff[4*i+0] = c_{i,0}, scoeff[4*i+1] = c_{i,1},           */
/* scoeff[4*i+2] = c_{i,2}, scoeff[4*i+3] = c_{i,3},           */
/* i=0,...,N-2   */
status = dfdConstruct1D( task, DF_PP_SPLINE, DF_METHOD_STD );

/* Check the Data Fitting operation status */
...

/* Initialize interpolation parameters */
nsite = NSITE;

/* Set site values */
...

sitehint = DF_NON_UNIFORM_PARTITION; /* Partition of sites is non-uniform */

/* Request to compute spline values */
ndorder = 1;
dorder = 1;
datahint = DF_NO_APRIORI_INFO; /* No additional information about breakpoints or
                               sites is provided. */
rhint = DF_MATRIX_STORAGE_ROWS; /* The library packs interpolation results
                                in row-major format. */
cell = NULL;                  /* Cell indices are not required. */

/* Solve interpolation problem using the default method: compute the spline values
   at the points site(i), i=0,..., nsite-1 and place the results to array r */
status = dfdInterpolate1D( task, DF_INTERP, DF_METHOD_STD, nsite, site,
                          sitehint, ndorder, &dorder, datahint, r, rhint, cell );

/* Check Data Fitting operation status */
...

/* De-allocate Data Fitting task resources */
status = dfDeleteTask( &task );
/* Check Data Fitting operation status */
...
return 0;
}

```

The following C example demonstrates how to compute indices of cells containing given sites. This example uses uniform partition presented with two boundary points. The sites are in the ascending order.

C Example of Cell Search

```

#include "mkl.h"

#define NX 100                      /* Size of partition, number of breakpoints */
#define NSITE 1000                   /* Number of interpolation sites */

int main()

```

```

{
    int status;          /* Status of a Data Fitting operation */
    DFTaskPtr task;     /* Data Fitting operations are task based */

    /* Parameters describing the partition */
    MKL_INT nx;          /* The size of partition x */
    float x[2];          /* Partition x is uniform and holds endpoints
                           of interpolation interval [a, b] */
    MKL_INT xhint;        /* Additional information about the structure of breakpoints */

    /* Parameters describing the function */
    MKL_INT ny;          /* Function dimension */
    float *y;             /* Function values at the breakpoints */
    MKL_INT yhint;        /* Additional information about the function */

    /* Parameters describing cell search */
    MKL_INT nsite;        /* Number of interpolation sites */
    float site[NSITE];   /* Array of interpolation sites */
    MKL_INT sitehint;     /* Additional information about the structure of sites */

    float* datahint;      /* Additional information on partition and interpolation sites */

    MKL_INT cell[NSITE]; /* Array for cell indices */

    /* Initialize a uniform partition */
    nx = NX;
    /* Set values of partition x: for uniform partition,           */
    /* provide end-points of the interpolation interval [-1.0,1.0] */
    x[0] = -1.0f; x[1] = 1.0f;
    xhint = DF_UNIFORM_PARTITION; /* Partition is uniform */

    /* Initialize function parameters */
    /* In cell search, function values are not necessary and are set to zero/NULL values */
    ny = 0;
    y = NULL;
    yhint = DF_NO_HINT;

    /* Create a Data Fitting task */
    status = dfsNewTask1D( &task, nx, x, xhint, ny, y, yhint );

    /* Check Data Fitting operation status */
    ...

    /* Initialize interpolation (cell search) parameters */
    nsite = NSITE;

    /* Set sites in the ascending order */
    ...
    sitehint = DF_SORTED_DATA; /* Sites are provided in the ascending order. */
    datahint = DF_NO_APRIORI_INFO; /* No additional information
                                   about breakpoints/sites is provided.*/

    /* Use a standard method to compute indices of the cells that contain
       interpolation sites. The library places the index of the cell containing
       site(i) to the cell(i), i=0,...,nsite-1 */
    status = dfsSearchCells1D( task, DF_METHOD_STD, nsite, site, sitehint,
                               datahint, cell );
    /* Check Data Fitting operation status */
    ...
}

```

```

/* Process cell indices */
...
/* Deallocate Data Fitting task resources */
status = dfDeleteTask( &task );

/* Check Data Fitting operation status */
...
return 0;
}

```

Task Status and Error Reporting

The Data Fitting routines report a task status through integer values. Negative status values indicate errors, while positive values indicate warnings. An error can be caused by invalid parameter values or a memory allocation failure.

The status codes have symbolic names predefined in the header file: for the C/C++ interface, these names are defined as macros via the `#define` statements; for the Fortran interface, the names are defined as integer constants via the `PARAMETER` operators.

If no error occurred, the function returns the `DF_STATUS_OK` code defined as zero:

C/C++:	<code>#define DF_STATUS_OK 0</code>
F90/F95:	<code>INTEGER, PARAMETER::DF_STATUS_OK = 0</code>

In case of an error, the function returns a non-zero error code that specifies the origin of the failure. Header files for both C/C++ and Fortran languages define the following status codes:

Status Codes in the Data Fitting Component

Status Code	Description
Common Status Codes	
<code>DF_STATUS_OK</code>	Operation completed successfully.
<code>DF_ERROR_NULL_TASK</code>	Data Fitting task is a <code>NULL</code> pointer.
<code>DF_ERROR_MEM_FAILURE</code>	Memory allocation failure.
<code>DF_ERROR_METHOD_NOT_SUPPORTED</code>	Requested method is not supported.
<code>DF_ERROR_COMP_TYPE_NOT_SUPPORTED</code>	Requested computation type is not supported.
<code>DF_ERROR_NULL_PTR</code>	Pointer to parameter is null.
Data Fitting Task Creation and Initialization, and Generic Editing Operations	
<code>DF_ERROR_BAD_NX</code>	Invalid number of breakpoints.
<code>DF_ERROR_BAD_X</code>	Array of breakpoints is invalid.
<code>DF_ERROR_BAD_X_HINT</code>	Invalid hint describing the structure of the partition.
<code>DF_ERROR_BAD_NY</code>	Invalid dimension of vector-valued function y .
<code>DF_ERROR_BAD_Y</code>	Array of function values is invalid.
<code>DF_ERROR_BAD_Y_HINT</code>	Invalid flag describing the structure of function y
Data Fitting Task-Specific Editing Operations	
<code>DF_ERROR_BAD_SPLINE_ORDER</code>	Invalid spline order.

Status Code	Description
DF_ERROR_BAD_SPLINE_TYPE	Invalid spline type.
DF_ERROR_BAD_IC_TYPE	Type of internal conditions used for spline construction is invalid.
DF_ERROR_BAD_IC	Array of internal conditions for spline construction is not defined.
DF_ERROR_BAD_BC_TYPE	Type of boundary conditions used in spline construction is invalid.
DF_ERROR_BAD_BC	Array of boundary conditions for spline construction is not defined.
DF_ERROR_BAD_PP_COEFF	Array of piecewise polynomial spline coefficients is not defined.
DF_ERROR_BAD_PP_COEFF_HINT	Invalid flag describing the structure of the piecewise polynomial spline coefficients.
DF_ERROR_BAD_PERIODIC_VAL	Function values at the endpoints of the interpolation interval are not equal as required in periodic boundary conditions.
DF_ERROR_BAD_DATA_ATTR	Invalid attribute of the pointer to be set or modified in Data Fitting task descriptor with the df?editidxptr task editor.
DF_ERROR_BAD_DATA_IDX	Index of the pointer to be set or modified in the Data Fitting task descriptor with the df?editidxptr task editor is out of the pre-defined range.

Data Fitting Computation Operations

DF_ERROR_BAD_NSITE	Invalid number of interpolation sites.
DF_ERROR_BAD_SITE	Array of interpolation sites is not defined.
DF_ERROR_BAD_SITE_HINT	Invalid flag describing the structure of interpolation sites.
DF_ERROR_BAD_NDORDER	Invalid size of the array defining derivative orders to be computed at interpolation sites.
DF_ERROR_BAD_DORDER	Array defining derivative orders to be computed at interpolation sites is not defined.
DF_ERROR_BAD_DATA_HINT	Invalid flag providing additional information about partition or interpolation sites.
DF_ERROR_BAD_INTERP	Array of spline-based interpolation results is not defined.
DF_ERROR_BAD_INTERP_HINT	Invalid flag defining the structure of spline-based interpolation results.
DF_ERROR_BAD_CELL_IDX	Array of indices of partition cells containing interpolation sites is not defined.
DF_ERROR_BAD_NLIM	Invalid size of arrays containing integration limits.
DF_ERROR_BAD_LLIM	Array of the left-side integration limits is not defined.

Status Code	Description
DF_ERROR_BAD_RLIM	Array of the right-side integration limits is not defined.
DF_ERROR_BAD_INTEGR	Array of spline-based integration results is not defined.
DF_ERROR_BAD_INTEGR_HINT	Invalid flag providing the structure of the array of spline-based integration results.
DF_ERROR_BAD_LOOKUP_INTERP_SITE	Bad site provided for interpolation with look-up interpolator.

NOTE

The routine that estimates piecewise polynomial cubic spline coefficients can return internal error codes related to the specifics of the implementation. Such error codes indicate invalid input data or other issues unrelated to Data Fitting routines.

Task Creation and Initialization Routines

Task creation and initialization routines are functions used to create a new task descriptor and initialize its parameters. The Data Fitting component provides the `df?newtask1d` routine that creates and initializes a new task descriptor for a one-dimensional Data Fitting task.

df?newtask1d

Creates and initializes a new task descriptor for a one-dimensional Data Fitting task.

Syntax

Fortran:

```
status = dfsnewtask1d(task, nx, x, xhint, ny, y, yhint)
status = dfdnewtask1d(task, nx, x, xhint, ny, y, yhint)
```

C:

```
status = dfsNewTask1D(&task, nx, x, xhint, ny, y, yhint)
status = dfdNewTask1D(&task, nx, x, xhint, ny, y, yhint)
```

Include Files

- Fortran: `mkl_df.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>nx</i>	Fortran: INTEGER C: const MKL_INT	Number of breakpoints representing partition of interpolation interval $[a, b]$.

Name	Type	Description
<i>x</i>	<p>Fortran: <code>REAL(KIND=4) DIMENSION(*)</code> <code>for dfsnewtask1d</code></p> <p>Fortran: <code>REAL(KIND=8) DIMENSION(*)</code> <code>for dfdnewtask1d</code></p> <p>C: <code>const float* for</code> <code>dfsNewTask1D</code></p> <p><code>const double* for</code> <code>dfdNewTask1D</code></p>	One-dimensional array containing the strictly sorted breakpoints from interpolation interval $[a, b]$. The structure of the array is defined by parameter <i>xhint</i> :
		<ul style="list-style-type: none"> If partition is non-uniform or quasi-uniform, the array should contain <i>nx</i> strictly ordered values. If partition is uniform, the array should contain two entries that represent endpoints of interpolation interval $[a, b]$.
<i>xhint</i>	<p>Fortran: INTEGER</p> <p>C: <code>const MKL_INT</code></p>	A flag describing the structure of partition <i>x</i> . For the list of possible values of <i>xhint</i> , see table " Hint Values for Partition <i>x</i> ". If you set the flag to the <code>DF_NO_HINT</code> value, the library interprets the partition as non-uniform.
<i>ny</i>	<p>Fortran: INTEGER</p> <p>C: <code>const MKL_INT</code></p>	Dimension of vector-valued function <i>y</i> .
<i>y</i>	<p>Fortran: <code>REAL(KIND=4) DIMENSION(*)</code> <code>for dfsnewtask1d</code></p> <p><code>REAL(KIND=8) DIMENSION(*)</code> <code>for dfdnewtask1d</code></p> <p>C: <code>const float* for</code> <code>dfsNewTask</code></p> <p><code>const double* for dfdNewTask</code></p>	Vector-valued function <i>y</i> , array of size <i>nx*ny</i> . The storage format of function values in the array is defined by the value of flag <i>yhint</i> .
<i>yhint</i>	<p>Fortran: INTEGER</p> <p>C: <code>const MKL_INT</code></p>	A flag describing the structure of array <i>y</i> . Valid hint values are listed in table " Hint Values for Vector-Valued Function <i>y</i> ". If you set the flag to the <code>DF_NO_HINT</code> value, the library assumes that all <i>ny</i> coordinates of the vector-valued function <i>y</i> are provided and stored in row-major format.

Output Parameters

Name	Type	Description
<i>task</i>	<p>Fortran: TYPE(DF_TASK)</p> <p>C: <code>DFTaskPtr</code></p>	Descriptor of the task.
<i>status</i>	<p>Fortran: INTEGER</p> <p>C: <code>int</code></p>	Status of the routine:
		<ul style="list-style-type: none"> <code>DF_STATUS_OK</code> if the task is created successfully. Non-zero error code if the task creation failed. See "Task Status and Error Reporting" for error code definitions.

Description

The `df?newtask1d` routine creates and initializes a new Data Fitting task descriptor with user-specified parameters for a one-dimensional Data Fitting task. The `x` and `nx` parameters representing the partition of interpolation interval $[a, b]$ are mandatory. If you provide invalid values for these parameters, such as a `NULL` pointer `x` or the number of breakpoints smaller than two, the routine does not create the Data Fitting task and returns an error code.

If you provide a vector-valued function `y`, make sure that the function dimension `ny` and the array of function values `y` are both valid. If any of these parameters are invalid, the routine does not create the Data Fitting task and returns an error code.

If you store coordinates of the vector-valued function `y` in non-contiguous memory locations, you can set the `yhint` flag to `DF_1ST_COORDINATE`, and pass only the first coordinate of the function into the task creation routine. After successful creation of the Data Fitting task, you can pass the remaining coordinates using the `df?editidxptr` task editor.

If the routine fails to create the task descriptor, it returns a `NULL` task pointer.

The routine supports the following hint values for partition `x`:

Hint Values for Partition `x`

Value	Description
<code>DF_NON_UNIFORM_PARTITION</code>	Partition is non-uniform.
<code>DF_QUASI_UNIFORM_PARTITION</code>	Partition is quasi-uniform.
<code>DF_UNIFORM_PARTITION</code>	Partition is uniform.
<code>DF_NO_HINT</code>	No hint is provided. By default, partition is interpreted as non-uniform.

The routine supports the following hint values for the vector-valued function:

Hint Values for Vector-Valued Function `y`

Value	Description
<code>DF_MATRIX_STORAGE_ROWS</code>	Data is stored in row-major format according to C conventions.
<code>DF_MATRIX_STORAGE_COLS</code>	Data is stored in column-major format according to Fortran conventions.
<code>DF_1ST_COORDINATE</code>	The first coordinate of vector-valued data is provided.
<code>DF_NO_HINT</code>	No hint is provided. By default, the coordinates of vector-valued function <code>y</code> are provided and stored in row-major format.

NOTE

You must preserve the arrays `x` (breakpoints) and `y` (vector-valued functions) through the entire workflow of the Data Fitting computations for a task, as the task stores the addresses of the arrays for spline-based computations.

Task Configuration Routines

In order to configure tasks, you can use task editors and task query routines.

Task editors initialize or change the predefined Data Fitting task parameters. You can use task editors to initialize or modify pointers to arrays or parameter values.

Task editors can be task-specific or generic. Task-specific editors can modify more than one parameter related to a specific task. Generic editors modify a single parameter at a time.

The Data Fitting component of Intel MKL provides the following task editors:

Data Fitting Task Editors

Editor	Description	Type
<code>df?editppsplin1d</code>	Changes parameters of the piecewise polynomial spline.	Task-specific
<code>df?editptr</code>	Changes a pointer in the task descriptor.	Generic
<code>dfieditval</code>	Changes a value in the task descriptor.	Generic
<code>df?editidxptr</code>	Changes a coordinate of data represented in matrix format, such as a vector-valued function or spline coefficients.	Generic

Task query routines are used to read the predefined Data Fitting task parameters. You can use task query routines to read the values of pointers or parameters.

Task query routines are generic (not task-specific), allowing you to read a single parameter at a time.

The Data Fitting component of the Intel MKL provides the following task query routines:

Data Fitting Task Query Routines

Editor	Description	Type
<code>df?queryptr</code>	Queries a pointer in the task descriptor.	Generic
<code>dfiqueryval</code>	Queries a value in the task descriptor.	Generic
<code>df?queryidxptr</code>	Queries a coordinate of data represented in matrix format, such as a vector-valued function or spline coefficients.	Generic

df?editppsplin1d

Modifies parameters representing a spline in a Data Fitting task descriptor.

Syntax

Fortran:

```
status = dfseditppsplin1d(task, s_order, s_type, bc_type, bc, ic_type, ic, scoeff, scoeffhint)
```

```
status = dfdeditppsplin1d(task, s_order, s_type, bc_type, bc, ic_type, ic, scoeff, scoeffhint)
```

C:

```
status = dfsEditPPSpline1D(task, s_order, s_type, bc_type, bc, ic_type, ic, scoeff, scoeffhint)
```

```
status = dfdEditPPSpline1D(task, s_order, s_type, bc_type, bc, ic_type, ic, scoeff, scoeffhint)
```

Include Files

- Fortran: mkl_df.f90
- C: mkl.h

Input Parameters

Name	Type	Description
task	<p>Fortran: TYPE(DF_TASK)</p> <p>C: DFTaskPtr</p>	Descriptor of the task.
s_order	<p>Fortran: INTEGER</p> <p>C: const MKL_INT</p>	Spline order. The parameter takes one of the values described in table " Spline Orders Supported by Data Fitting Functions ".
s_type	<p>Fortran: INTEGER</p> <p>C: const MKL_INT</p>	Spline type. The parameter takes one of the values described in table " Spline Types Supported by Data Fitting Functions ".
bc_type	<p>Fortran: INTEGER</p> <p>C: const MKL_INT</p>	Type of boundary conditions. The parameter takes one of the values described in table " Boundary Conditions Supported by Data Fitting Functions ".
bc	<pre> Fortran: REAL(KIND=4) DIMENSION(*) for dfreditppsplinelD REAL(KIND=8) DIMENSION(*) for dfdeditppsplinelD C: const float* for dfsEditPPSpline1D const double* for dfdEditPPSpline1D </pre>	Pointer to boundary conditions. The size of the array is defined by the value of parameter <i>bc_type</i> : <ul style="list-style-type: none"> • If you set free-end or not-a-knot boundary conditions, pass the NULL pointer to this parameter. • If you combine boundary conditions at the endpoints of the interpolation interval, pass an array of two elements. • If you set a boundary condition for the default quadratic spline or a periodic condition for Hermite or the default cubic spline, pass an array of one element.
ic_type	<p>Fortran: INTEGER</p> <p>C: const MKL_INT</p>	Type of internal conditions. The parameter takes one of the values described in table " Internal Conditions Supported by Data Fitting Functions ".
ic	<pre> Fortran: REAL(KIND=4) DIMENSION(*) for dfreditppsplinelD REAL(KIND=8) DIMENSION(*) for dfdeditppsplinelD C: const float* for dfsEditPPSpline1D const double* for dfdEditPPSpline1D </pre>	A non-NULL pointer to the array of internal conditions. The size of the array is defined by the value of parameter <i>ic_type</i> : <ul style="list-style-type: none"> • If you set first derivatives or second derivatives internal conditions (<i>ic_type</i>=DF_IC_1ST_DER or <i>ic_type</i>=DF_IC_2ND_DER), pass an array of n-1 derivative values at the internal points of the interpolation interval. • If you set the knot values internal condition for Subbotin spline (<i>ic_type</i>=DF_IC_Q_KNOT) and the knot partition is non-uniform, pass an array of n+1 elements. • If you set the knot values internal condition for Subbotin spline (<i>ic_type</i>=DF_IC_Q_KNOT) and the knot partition is uniform, pass an array of four elements.

Name	Type	Description
<i>scoeff</i>	Fortran: REAL(KIND=4) DIMENSION(*) for <code>dfseditppsplin1D</code> REAL(KIND=8) DIMENSION(*) for <code>dfdeditppsplin1D</code> C: const float* for <code>dfsEditPPSpline1D</code> const double* for <code>dfdEditPPSpline1D</code>	Spline coefficients. An array of size $s_order * (nx-1)$. The storage format of the coefficients in the array is defined by the value of flag <i>scoeffhint</i> .
<i>scoeffhint</i>	Fortran: INTEGER C: const MKL_INT	A flag describing the structure of the array of spline coefficients. For valid hint values, see table " Hint Values for Spline Coefficients ". The library stores the coefficients in row-major format. The default value is <code>DF_NO_HINT</code> .

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.

Description

The editor modifies parameters that describe the order, type, boundary conditions, internal conditions, and coefficients of a spline. The spline order definition is provided in the "[Mathematical Conventions](#)" section. You can set the spline order to any value supported by Data Fitting functions. The table below lists the available values:

Spline Orders Supported by the Data Fitting Functions

Order	Description
<code>DF_PP_STD</code>	Artificial value. Use this value for look-up and step-wise constant interpolants only.
<code>DF_PP_LINEAR</code>	Piecewise polynomial spline of the second order (linear spline).
<code>DF_PP_QUADRATIC</code>	Piecewise polynomial spline of the third order (quadratic spline).
<code>DF_PP_CUBIC</code>	Piecewise polynomial spline of the fourth order (cubic spline).

To perform computations with a spline not supported by Data Fitting routines, set the parameter defining the spline order and pass the spline coefficients to the library in the supported format. For format description, see figure "[Row-major Coefficient Storage Format](#)".

The table below lists the supported spline types:

Spline Types Supported by Data Fitting Functions

Type	Description
DF_PP_DEFAULT	The default spline type. You can use this type with linear, quadratic, or user-defined splines.
DF_PP_SUBBOTIN	Quadratic splines based on Subbotin algorithm, [StechSub76].
DF_PP_NATURAL	Natural cubic spline.
DF_PP_HERMITE	Hermite cubic spline.
DF_PP_BESSEL	Bessel cubic spline.
DF_PP_AKIMA	Akima cubic spline.
DF_LOOKUP_INTERPOLANT	Look-up interpolant.
DF_CR_STEPWISE_CONST_INTERPOLANT	Continuous right step-wise constant interpolant.
DF_CL_STEPWISE_CONST_INTERPOLANT	Continuous left step-wise constant interpolant.

If you perform computations with look-up or step-wise constant interpolants, set the spline order to the DF_PP_STD value.

Construction of specific splines may require boundary or internal conditions. To compute coefficients of such splines, you should pass boundary or internal conditions to the library by specifying the type of the conditions and providing the necessary values. For splines that do not require additional conditions, such as linear splines, set condition types to DF_NO_BC and DF_NO_IC, and pass NULL pointers to the conditions. The table below defines the supported boundary conditions:

Boundary Conditions Supported by Data Fitting Functions

Boundary Condition	Description	Spline
DF_NO_BC	No boundary conditions provided.	All
DF_BC_NOT_A_KNOT	Not-a-knot boundary conditions.	Akima, Bessel, Hermite, natural cubic
DF_BC_FREE_END	Free-end boundary conditions.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_1ST_LEFT_DER	The first derivative at the left endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_1ST_RIGHT_DER	The first derivative at the right endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_2ST_LEFT_DER	The second derivative at the left endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_2ND_RIGHT_DER	The second derivative at the right endpoint.	Akima, Bessel, Hermite, natural cubic, quadratic Subbotin
DF_BC_PERIODIC	Periodic boundary conditions.	Linear, all cubic splines
DF_BC_Q_VAL	Function value at point $(x_0 + x_1)/2$	Default quadratic

NOTE

To construct a natural cubic spline, pass these settings to the editor:

- `DF_PP_CUBIC` as the spline order,
- `DF_PP_NATURAL` as the spline type, and
- `DF_BC_FREE_END` as the boundary condition.

To construct a cubic spline with other boundary conditions, pass these settings to the editor:

- `DF_PP_CUBIC` as the spline order,
- `DF_PP_NATURAL` as the spline type, and
- the required type of boundary condition.

For Akima, Hermite, Bessel and default cubic splines use the corresponding type defined in [Table Spline Types Supported by Data Fitting Functions](#).

You can combine the values of boundary conditions with a bitwise OR operation. This permits you to pass combinations of first and second derivatives at the endpoints of the interpolation interval into the library. To pass a first derivative at the left endpoint and a second derivative at the right endpoint, set the boundary conditions to `DF_BC_1ST_LEFT_DER` OR `DF_BC_2ND_RIGHT_DER`.

You should pass the combined boundary conditions as an array of two elements. The first entry of the array contains the value of the boundary condition for the left endpoint of the interpolation interval, and the second entry - for the right endpoint. Pass other boundary conditions as arrays of one element.

For the conditions defined as a combination of valid values, the library applies the following rules to identify the boundary condition type:

- If not required for spline construction, the value of boundary conditions is ignored.
- Not-a-knot condition has the highest priority. If set, other boundary conditions are ignored.
- Free-end condition has the second priority after the not-a-knot condition. If set, other boundary conditions are ignored.
- Periodic boundary condition has the next priority after the free-end condition.
- The first derivative has higher priority than the second derivative at the right and left endpoints.

If you set the periodic boundary condition, make sure that function values at the endpoints of the interpolation interval are identical. Otherwise, the library returns an error code. The table below specifies the values to be provided for each type of spline if the periodic boundary condition is set.

Boundary Requirements for Periodic Conditions

Spline Type	Periodic Boundary Condition Support	Boundary Value
Linear	Yes	Not required
Default quadratic	No	
Subbotin quadratic	No	
Natural cubic	Yes	Not required
Bessel	Yes	Not required
Akima	Yes	Not required
Hermite cubic	Yes	First derivative
Default cubic	Yes	Second derivative

Internal conditions supported in the Data Fitting domain that you can use for the `ic_type` parameter are the following:

Internal Conditions Supported by Data Fitting Functions

Internal Condition	Description	Spline
DF_NO_IC	No internal conditions provided.	
DF_IC_1ST_DER	Array of first derivatives of size $n-2$, where n is the number of breakpoints. Derivatives are applicable to each coordinate of the vector-valued function.	Hermite cubic
DF_IC_2ND_DER	Array of second derivatives of size $n-2$, where n is the number of breakpoints. Derivatives are applicable to each coordinate of the vector-valued function.	Default cubic
DF_IC_Q_KNOT	Knot array of size $n+1$, where n is the number of breakpoints.	Subbotin quadratic

To construct a Subbotin quadratic spline, you have three options to get the array of knots in the library:

- If you do not provide the knots, the library uses the default values of knots $t = \{t_i\}$, $i = 0, \dots, n$ according to the rule:

$$t_0 = x_0, t_n = x_{n-1}, t_i = (x_i + x_{i-1})/2, i = 1, \dots, n - 1.$$
- If you provide the knots in an array of size $n + 1$, the knots form a non-uniform partition. Make sure that the knot values you provide meet the following conditions:

$$t_0 = x_0, t_n = x_{n-1}, t_i \in (x_{i-1}, x_i), i = 1, \dots, n - 1.$$
- If you provide the knots in an array of size 4, the knots form a uniform partition

$$t_0 = x_0, t_1 = l, t_2 = r, t_3 = x_{n-1}, \text{ where } l \in (x_0, x_1) \text{ and } r \in (x_{n-2}, x_{n-1}).$$

In this case, you need to set the value of the `ic_type` parameter holding the type of internal conditions to `DF_IC_Q_KNOT` OR `DF_UNIFORM_PARTITION`.

NOTE

Since the partition is uniform, perform an OR operation with the `DF_UNIFORM_PARTITION` partition hint value described in [Table Hint Values for Partition x](#).

For computations based on look-up and step-wise constant interpolants, you can avoid calling the `df?editppsspline1d` editor and directly call one of the routines for spline-based computation of spline values, derivatives, or integrals. For example, you can call the `df?construct1d` routine to construct the required spline with the given attributes, such as order or type.

The memory location of the spline coefficients is defined by the `scoeff` parameter. Make sure that the size of the array is sufficient to hold `s_order * (nx-1)` values.

The `df?editppsspline1d` routine supports the following hint values for spline coefficients:

Hint Values for Spline Coefficients

Order	Description
DF_1ST_COORDINATE	The first coordinate of vector-valued data is provided.

Order	Description
DF_NO_HINT	No hint is provided. By default, all sets of spline coefficients are stored in row-major format.

The coefficients for all coordinates of the vector-valued function are packed in memory one by one in successive order, from function y_1 to function y_n .

Within each coordinate, the library stores the coefficients as an array, in row-major format:

$$c_{1,0}, c_{1,1}, \dots, c_{1,k}, c_{2,0}, c_{2,1}, \dots, c_{2,k}, \dots, c_{n-1,0}, c_{n-1,1}, \dots, c_{n-1,k}$$

Mapping of the coefficients to storage in the *scoeff* array is described below, where $c_{i,j}$ is the j th coefficient of the function

$$P_i(x) = c_{i,0} + c_{i,1}(x - x_i) + \dots + c_{i,k}(x - x_i)^k$$

See [Mathematical Conventions](#) for more details on nomenclature and interpolants.

Row-major Coefficient Storage Format

$$\begin{aligned} P_1(x) &= \overrightarrow{c_{1,0}} + \overrightarrow{c_{1,1}(x - x_1)} + \cdots + \overrightarrow{c_{1,k}(x - x_1)^k} \\ P_2(x) &= \overleftarrow{c_{2,0}} + \overrightarrow{c_{2,1}(x - x_2)} + \cdots + \overrightarrow{c_{2,k}(x - x_2)^k} \\ &\vdots & \vdots & \vdots \\ P_{n-1}(x) &= \overleftarrow{c_{n-1,0}} + \overrightarrow{c_{n-1,1}(x - x_{n-1})} + \cdots + \overrightarrow{c_{n-1,k}(x - x_{n-1})^k} \end{aligned}$$

If you store splines corresponding to different coordinates of the vector-valued function at non-contiguous memory locations, do the following:

1. Set the *scoeffhint* flag to DF_1ST_COORDINATE and provide the spline for the first coordinate.
2. Pass the spline coefficients for the remaining coordinates into the Data Fitting task using the df?editidxptra task editor.

Using the df?editppspine1d task editor, you can provide to the Data Fitting task an already constructed spline that you want to use in computations. To ensure correct interpretation of the memory content, you should set the following parameters:

- Spline order and type, if appropriate. If the spline is not supported by the library, set the *s_type* parameter to DF_PP_DEFAULT.
- Pointer to the array of spline coefficients in row-major format.
- The *scoeffhint* parameter describing the structure of the array:
 - Set the *scoeffhint* flag to the DF_1ST_COORDINATE value to pass spline coefficients stored at different memory locations. In this case, you can set the parameters that describe boundary and internal conditions to zero.

- Use the default value `DF_NO_HINT` for all other cases.

Before passing an already constructed spline into the library, you should call the `dfieditval` task editor to provide the dimension of the spline `DF_NY`. See table "[Parameters Supported by the dfieditval Task Editor](#)" for details.

After you provide the spline to the Data Fitting task, you can run computations that use this spline.

NOTE

You must preserve the arrays `bc` (boundary conditions), `ic` (internal conditions), and `scoeff` (spline coefficients) through the entire workflow of the Data Fitting computations for a task, as the task stores the addresses of the arrays for spline-based computations.

df?editptr

Modifies a pointer to an array held in a Data Fitting task descriptor.

Syntax

Fortran:

```
status = dfseditptr(task, ptr_attr, ptr)
status = dfdeditptr(task, ptr_attr, ptr)
```

C:

```
status = dfsEditPtr(task, ptr_attr, ptr)
status = dfdEditPtr(task, ptr_attr, ptr)
```

Include Files

- Fortran: `mkl_df.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<code>task</code>	Fortran: <code>TYPE(DF_TASK)</code> C: <code>DFTaskPtr</code>	Descriptor of the task.
<code>ptr_attr</code>	Fortran: <code>INTEGER</code> C: <code>const MKL_INT</code>	The parameter to change. For details, see the <i>Pointer Attribute</i> column in table " Pointers Supported by the df?editptr Task Editor ".
<code>ptr</code>	Fortran: <code>REAL(KIND=4) DIMENSION(*)</code> <code>for dfseditptr</code> <code>REAL(KIND=8) DIMENSION(*)</code> <code>for dfdeditptr</code> C: <code>const float* for</code> <code>dfsEditPtr</code> <code>const double* for dfdEditPtr</code>	New pointer. For details, see the <i>Purpose</i> column in table " Pointers Supported by the df?editptr Task Editor ".

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	<p>Status of the routine:</p> <ul style="list-style-type: none"> • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The `df?editptr` editor replaces the pointer of type `ptr_attr` stored in a Data Fitting task descriptor with a new pointer `ptr`. The table below describes types of pointers supported by the editor:

Pointers Supported by the `df?editptr` Task Editor

Pointer Attribute	Purpose
<code>DF_X</code>	Partition <code>x</code> of the interpolation interval, an array of strictly sorted breakpoints
<code>DF_Y</code>	Vector-valued function <code>y</code>
<code>DF_IC</code>	Internal conditions for spline construction. For details, see table " Internal Conditions Supported by Data Fitting Functions ".
<code>DF_BC</code>	Boundary conditions for spline construction. For details, see table " Boundary Conditions Supported by Data Fitting Functions ".
<code>DF_PP_SCOEFF</code>	Spline coefficients

You can use `df?editptr` to modify different types of pointers including pointers to the vector-valued function and spline coefficients stored in contiguous memory. Use the `df?editidxptr` editor if you need to modify pointers to coordinates of the vector-valued function or spline coefficients stored at non-contiguous memory locations.

If you modify a partition of the interpolation interval, then you should call the `dfieditval` task editor with the corresponding value of `DF_XHINT`, even if the structure of the partition remains the same.

If you pass a `NULL` pointer to the `df?editptr` task editor, the task remains unchanged and the routine returns an error code. For the predefined error codes, please see "[Task Status and Error Reporting](#)".

NOTE

You must preserve the arrays `x` (breakpoints), `y` (vector-valued functions), `bc` (boundary conditions), `ic` (internal conditions), and `scoeff` (spline coefficients) through the entire workflow of the Data Fitting computations which use those arrays, as the task stores the addresses of the arrays for spline-based computations.

`dfieditval`

Modifies a parameter value in a Data Fitting task descriptor.

Syntax

Fortran:

```
status = dfieditval(task, val_attr, val)
```

C:

```
status = dfiEditVal(task, val_attr, val)
```

Include Files

- Fortran: mkl_df.f90
- C: mkl.h

Input Parameters

Name	Type	Description
task	Fortran: TYPE(DF_TASK) C: DFTaskPtr	Descriptor of the task.
val_attr	Fortran: INTEGER C: const MKL_INT	The parameter to change. See table " Parameters Supported by the dfieditval Task Editor ".
val	Fortran: INTEGER C: const MKL_INT	A new parameter value. See table " Parameters Supported by the dfieditval Task Editor ".

Output Parameters

Name	Type	Description
status	Fortran: INTEGER C: int	Status of the routine: <ul style="list-style-type: none"> • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The `dfieditval` task editor replaces the parameter of type `val_attr` stored in a Data Fitting task descriptor with a new value `val`. The table below describes valid types of parameter `val_attr` supported by the editor:

Parameters Supported by the dfieditval Task Editor

Parameter Attribute	Purpose
DF_NX	Number of breakpoints
DF_XHINT	A flag describing the structure of partition. See table " Hint Values for Partition x " for the list of available values.
DF_NY	Dimension of the vector-valued function
DF_YHINT	A flag describing the structure of the vector-valued function. See table " Hint Values for Vector Function y " for the list of available values.
DF_SPLINE_ORDER	Spline order. See table " Spline Orders Supported by Data Fitting Functions " for the list of available values.

Parameter Attribute	Purpose
DF_SPLINE_TYPE	Spline type. See table " Spline Types Supported by Data Fitting Functions " for the list of available values.
DF_BC_TYPE	Type of boundary conditions used in spline construction. See table " Boundary Conditions Supported by Data Fitting Functions " for the list of available values.
DF_IC_TYPE	Type of internal conditions used in spline construction. See table " Internal Conditions Supported by Data Fitting Functions " for the list of available values.
DF_PP_COEFF_HINT	A flag describing the structure of spline coefficients. See table " Hint Values for Spline Coefficients " for the list of available values.
DF_CHECK_FLAG	A flag which controls checking of Data Fitting parameters. See table " Possible Values for the DF_CHECK_FLAG Parameter " for the list of available values.

If you pass a zero value for the parameter describing the size of the arrays that hold coefficients for a partition, a vector-valued function, or a spline, the parameter held in the Data fitting task remains unchanged and the routine returns an error code. For the predefined error codes, see "[Task Status and Error Reporting](#)".

[Possible Values for the DF_CHECK_FLAG Parameter](#)

Value	Description
DF_ENABLE_CHECK_FLAG	Checks the correctness of parameters of Data Fitting computational routines (default mode).
DF_DISABLE_CHECK_FLAG	Disables checking of the correctness of parameters of Data Fitting computational routines.

Use `DF_CHECK_FLAG` for `val_attr` in order to control validation of parameters of Data Fitting computational routines such as `df construct1d`, `df interpolate1d` `df interpolateex1d`, and `df searchcells1d` `df searchcellsex1d`, which can perform better with a small number of interpolation sites or integration limits (fewer than one dozen). The default mode, with checking of parameters enabled, should be used as you develop a Data Fitting-based application. After you complete development you can disable parameter checking in order to improve the performance of your application.

If you modify the parameter describing dimensions of the arrays that hold the vector-valued function or spline coefficients in contiguous memory, you should call the `df?editptr` task editor with the corresponding pointers to the vector-valued function or spline coefficients even when this pointer remains unchanged. Call the `df?editidxptr` editor if those arrays are stored in non-contiguous memory locations.

You must call the `dfieditidxptr` task editor to edit the structure of the partition `DF_XHINT` every time you modify a partition using `df editptr`, even if the structure of the partition remains the same.

[df?editidxptr](#)

Modifies a pointer to the memory representing a coordinate of the data stored in matrix format.

Syntax

Fortran:

```
status = dfseditidxptr(task, ptr_attr, idx, ptr)
status = dfdeditidxptr(task, ptr_attr, idx, ptr)
```

C:

```
status = dfsEditIdxPtr(task, ptr_attr, idx, ptr)
status = dfdEditIdxPtr(task, ptr_attr, idx, ptr)
```

Include Files

- Fortran: mkl_df.f90
- C: mkl.h

Input Parameters

Name	Type	Description
task	Fortran: TYPE(DF_TASK) C: DFTaskPtr	Descriptor of the task.
ptr_attr	Fortran: INTEGER C: const MKL_INT	Type of the data to be modified. The parameter takes one of the values described in "Data Attributes Supported by the df?editidxptr Task Editor" .
idx	Fortran: INTEGER C: const MKL_INT	Index of the coordinate whose pointer is to be modified.
ptr	Fortran: REAL(KIND=4) DIMENSION(*) for dfreditidxptr REAL(KIND=8) DIMENSION(*) for dfdeditidxptr C: const float* for dfsEditIdxPtr const double* for dfdEditIdxPtr	Pointer to the data that holds values of coordinate <i>idx</i> . For details, see table "Data Attributes Supported by the df?editidxptr Task Editor" .

Output Parameters

Name	Type	Description
status	Fortran: INTEGER C: int	Status of the routine: <ul style="list-style-type: none"> • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The routine modifies a pointer to the array that holds the *idx* coordinate of vector-valued function *y* or the pointer to the array of spline coefficients corresponding to the given coordinate.

You can use the editor if you need to pass into a Data Fitting task or modify the pointer to coordinates of the vector-valued function or spline coefficients held at non-contiguous memory locations. Do not use the editor for coordinates at contiguous memory locations in row-major format.

Before calling this editor, make sure that you have created and initialized the task using a task creation function or a relevant editor such as the generic or specific `df?editpppline1d` editor.

Data Attributes Supported by the df?editidxptr Task Editor

Data Attribute	Description
DF_Y	Vector-valued function y
DF_PP_SCOEFF	Piecewise polynomial spline coefficients

When using `df?editidxptr`, you might receive an error code in the following cases:

- You passed an unsupported parameter value into the editor.
- The value of the index exceeds the predefined value that equals the dimension `ny` of the vector-valued function.
- You pass a `NULL` pointer to the editor. In this case, the task remains unchanged.
- You pass a pointer to the `idx` coordinate of the vector-valued function you provided to contiguous memory in column-major format.

The code example below demonstrates how to use the editor for providing values of a vector-valued function stored in two non-contiguous arrays:

```
#define NX 1000 /* number of break points */
#define NY 2      /* dimension of vector-valued function */
int main()
{
    DFTaskPtr task;
    double x[NX];
    double y1[NX], y2[NX]; /* vector-valued function is stored as two arrays */
    /* Provide first coordinate of two-dimensional function y into creation routine */
    status = dfdNewTaskID( &task, NX, x, DF_NON_UNIFORM_PARTITION, NY, y1,
                           DF_1ST_COORDINATE );
    /* Provide second coordinate of two-dimensional function */
    status = dfdEditIdxPtr(task, DF_Y, 1, y2 );
    ...
}
```

df?queryptr

Reads a pointer to an array held in a Data Fitting task descriptor.

Syntax

Fortran:

```
status = dfsqueryptr(task, ptr_attr, ptr)
status = dfdqueryptr(task, ptr_attr, ptr)
```

C:

```
status = dfsQueryPtr(task, ptr_attr, ptr)
status = dfdQueryPtr(task, ptr_attr, ptr)
```

Include Files

- Fortran: `mkl_df.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(DF_TASK) C: DFTaskPtr	Descriptor of the task.
<i>ptr_attr</i>	Fortran: INTEGER C: const MKL_INT	The parameter to query. The query routine supports pointer attributes described in the table " Pointers Supported by the df?editptr Task Editor ". For details, see the <i>Pointer Attribute</i> column in the table.

Output Parameters

Name	Type	Description
<i>ptr</i>	Fortran: INTEGER(KIND=8) C: float** for dfsQueryPtr double** for dfdQueryPtr	Pointer to array returned by the query routine. For details, see the <i>Purpose</i> column in table " Pointers Supported by the df?editptr Task Editor ".
<i>status</i>	Fortran: INTEGER C: int	Status of the routine: <ul style="list-style-type: none"> • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The `df?queryptr` routine returns the pointer of type `ptr_attr` stored in a Data Fitting task descriptor as parameter `ptr`. Attributes of the pointers supported by the query function are identical to those supported by the editor `df?editptr` editor in the table "[Pointers Supported by the df?editptr Task Editor](#)".

You can use `df?queryptr` to read different types of pointers including pointers to the vector-valued function and spline coefficients stored in contiguous memory.

dfiqueryval

Reads a parameter value in a Data Fitting task descriptor.

Syntax

Fortran:

```
status = dfiqueryval(task, val_attr, val)
```

C:

```
status = dfiQueryVal(task, val_attr, val)
```

Include Files

- Fortran: `mkl_df.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(DF_TASK) C: DFTaskPtr	Descriptor of the task.
<i>val_attr</i>	Fortran: INTEGER C: const MKL_INT	The parameter to query. The query function supports the parameter attributes described in " Parameters Supported by the dfieditval Task Editor ".

Output Parameters

Name	Type	Description
<i>val</i>	Fortran: INTEGER C: MKL_INT	The parameter value returned by the query function. See table " Parameters Supported by the dfieditval Task Editor ".
<i>status</i>	Fortran: INTEGER C: int	Status of the routine: <ul style="list-style-type: none"> • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The dfiqueryval routine returns a parameter of type *val_attr* stored in a Data Fitting task descriptor as parameter *val*. The query function supports the parameter attributes described in "[Parameters Supported by the dfieditval Task Editor](#)".

df?queryidxptr

Reads a pointer to the memory representing a coordinate of the data stored in matrix format.

Syntax

Fortran:

```
status = dfsqueryidxptr(task, ptr_attr, idx, ptr)
status = dfdqueryidxptr(task, ptr_attr, idx, ptr)
```

C:

```
status = dfsQueryIdxPtr(task, ptr_attr, idx, ptr)
status = dfdQueryIdxPtr(task, ptr_attr, idx, ptr)
```

Include Files

- Fortran: mkl_df.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(DF_TASK) C: DFTaskPtr	Descriptor of the task.
<i>ptr_attr</i>	Fortran: INTEGER C: const MKL_INT	Pointer attribute to query. The parameter takes one of the attributes described in "Data Attributes Supported by the df?editidxptr Task Editor" .
<i>idx</i>	Fortran: INTEGER C: const MKL_INT	Index of the coordinate of the pointer to query.

Output Parameters

Name	Type	Description
<i>ptr</i>	Fortran: INTEGER(KIND=8) C: float* for dfsQueryIdxPtr double* for dfdQueryIdxPtr	Pointer to the data that holds values of coordinate <i>idx</i> returned. For details, see table "Data Attributes Supported by the df?editidxptr Task Editor" .
<i>status</i>	Fortran: INTEGER C: int	Status of the routine: <ul style="list-style-type: none"> • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

Description

The routine returns a pointer to the array that holds the *idx* coordinate of vector-valued function *y* or the pointer to the array of spline coefficients corresponding to the given coordinate.

You can use the query routine if you need the pointer to coordinates of the vector-valued function or spline coefficients held at non-contiguous memory locations or at a contiguous memory location in row-major format (the default storage format for spline coefficients).

Before calling this query routine, make sure that you have created and initialized the task using a task creation function or a relevant editor such as the generic or specific `df?editppspline1d` editor.

When using `df?queryidxptr`, you might receive an error code in the following cases:

- You passed an unsupported parameter value into the editor.
- The value of the index exceeds the predefined value that equals the dimension *ny* of the vector-valued function.
- You request the pointer to the *idx* coordinate of the vector-valued function you provided to contiguous memory in column-major format.

Computational Routines

Data Fitting computational routines are functions used to perform spline-based computations, such as:

- spline construction
- computation of values, derivatives, and integrals of the predefined order
- cell search

Once you create a Data Fitting task and initialize the required parameters, you can call computational routines as many times as necessary.

The table below lists the available computational routines:

Data Fitting Computational Routines

Routine	Description
<code>df?construct1d</code>	Constructs a spline for a one-dimensional Data Fitting task.
<code>df?interpolate1d</code>	Computes spline values and derivatives.
<code>df?interpolateex1d</code>	Computes spline values and derivatives by calling user-provided interpolants.
<code>df?integrate1d</code>	Computes spline-based integrals.
<code>df?integrateex1d</code>	Computes spline-based integrals by calling user-provided integrators.
<code>df?searchcells1d</code>	Finds indices of cells containing interpolation sites.
<code>df?searchcellsex1d</code>	Finds indices of cells containing interpolation sites by calling user-provided cell searchers.

If a Data Fitting computation completes successfully, the computational routines return the `DF_STATUS_OK` code. If an error occurs, the routines return an error code specifying the origin of the failure. Some possible errors are the following:

- The task pointer is `NULL`.
- Memory allocation failed.
- The computation failed for another reason.

For the list of available status codes, see "[Task Status and Error Reporting](#)".

NOTE

Data Fitting computational routines do not control errors for floating-point conditions, such as overflow, gradual underflow, or operations with Not a Number (NaN) values.

df?construct1d

Constructs a spline of the given type.

Syntax

Fortran:

```
status = dfsconstruct1d(task, s_format, method)
status = dfdconstruct1d(task, s_format, method)
```

C:

```
status = dfsConstruct1D(task, s_format, method)
status = dfdConstruct1D(task, s_format, method)
```

Include Files

- Fortran: `mkl_df.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(DF_TASK) C: DFTaskPtr	Descriptor of the task.
<i>s_format</i>	Fortran: INTEGER C: const MKL_INT	Spline format. The supported value is DF_PP_SPLINE.
<i>method</i>	Fortran: INTEGER C: const MKL_INT	Construction method. The supported value is DF_METHOD_STD.

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Status of the routine: <ul style="list-style-type: none"> • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.

Description

Before calling `df?construct1d`, you need to create and initialize the task, and set the parameters representing the spline. Then you can call the `df?construct1d` routine to construct the spline. The format of the spline is defined by parameter *s_format*. The method for spline construction is defined by parameter *method*. Upon successful construction, the spline coefficients are available in the user-provided memory location in the format you set through the Data Fitting editor. For the available storage formats, see table "[Hint Values for Spline Coefficients](#)".

df?interpolate1d/df?interpolateex1d

Runs data fitting computations.

Syntax

Fortran:

```
status = dfsinterpolate1d(task, type, method, nsite, site, sitehint, ndorder, dorder,
                          datahint, r, rhint, cell)

status = dfdinterpolate1d(task, type, method, nsite, site, sitehint, ndorder, dorder,
                          datahint, r, rhint, cell)

status = dfsinterpolateex1d(task, type, method, nsite, site, sitehint, ndorder,
                           dorder, datahint, r, rhint, cell, le_cb, le_params, re_cb, re_params,
                           i_cb, i_params, search_cb, search_params)

status = dfdinterpolateex1d(task, type, method, nsite, site, sitehint, ndorder,
                           dorder, datahint, r, rhint, cell, le_cb, le_params, re_cb, re_params,
                           i_cb, i_params, search_cb, search_params)
```

C:

```
status = dfsInterpolate1D(task, type, method, nsite, site, sitehint, ndorder, dorder,
                          datahint, r, rhint, cell)
```

```

status = dfdInterpolate1D(task, type, method, nsite, site, sitehint, ndorder, dorder,
                           datahint, r, rhint, cell)

status = dfsInterpolateEx1D(task, type, method, nsite, site, sitehint, ndorder,
                           dorder, datahint, r, rhint, cell, le_cb, le_params, re_cb, re_params, i_cb, i_params,
                           search_cb, search_params)

status = dfdInterpolateEx1D(task, type, method, nsite, site, sitehint, ndorder,
                           dorder, datahint, r, rhint, cell, le_cb, le_params, re_cb, re_params, i_cb, i_params,
                           search_cb, search_params)

```

Include Files

- Fortran: mkl_df.f90
- C: mkl.h

Input Parameters

Name	Type	Description
task	Fortran: TYPE(DF_TASK) C: DFTaskPtr	Descriptor of the task.
type	Fortran: INTEGER C: const MKL_INT	Type of spline-based computations. The parameter takes one or more values combined with an OR operation. For the list of possible values, see table "Computation Types Supported by the df?interpolate1d/ df?interpolate1d Routines".
method	Fortran: INTEGER C: const MKL_INT	Computation method. The supported value is DF_METHOD_PP.
nsite	Fortran: INTEGER C: const MKL_INT	Number of interpolation sites.
site	Fortran: REAL(KIND=4) DIMENSION(*) for dfsinterpolate1d/ dfsinterpolateex1d C: const float* for dfsInterpolate1D/ dfsInterpolateEx1D const double* for dfdInterpolate1D/ dfdInterpolateEx1D	Array of interpolation sites of size nsite. The structure of the array is defined by the sitehint parameter: <ul style="list-style-type: none"> If sites form a non-uniform partition, the array should contain nsite values. If sites form a uniform partition, the array should contain two entries that represent the left and the right interpolation sites. The first entry of the array contains the left-most interpolation point. The second entry of the array contains the right-most interpolation point.
sitehint	Fortran: INTEGER C: const MKL_INT	A flag describing the structure of the interpolation sites. For the list of possible values of sitehint, see table "Hint Values for Interpolation Sites". If you set the flag to DF_NO_HINT, the library interprets the site-defined partition as non-uniform.

Name	Type	Description
<i>ndorder</i>	Fortran: INTEGER C: const MKL_INT	Maximal derivative order increased by one to be computed at interpolation sites.
<i>dorder</i>	Fortran: INTEGER DIMENSION(*) C: const MKL_INT*	Array of size <i>ndorder</i> that defines the order of the derivatives to be computed at the interpolation sites. If all the elements in <i>dorder</i> are zero, the library computes the spline values only. If you do not need interpolation computations, set <i>ndorder</i> to zero and pass a NULL pointer to <i>dorder</i> .
<i>datahint</i>	Fortran: REAL(KIND=4) DIMENSION(*) for dfsinterpolate1d/ dfsinterpolateex1d REAL(KIND=8) DIMENSION(*) for dfdinterpolate1d/ dfdinterpolateex1d C: const float* for dfsInterpolate1D/ dfsInterpolateEx1D const double* for dfdInterpolate1D/ dfdInterpolateEx1D	Array that contains additional information about the structure of partition <i>x</i> and interpolation sites. This data helps to speed up the computation. If you provide a NULL pointer, the routine uses the default settings for computations. For details on the <i>datahint</i> array, see table "Structure of the <i>datahint</i> Array".
<i>r</i>	Fortran: REAL(KIND=4) DIMENSION(*) for dfsinterpolate1d/ dfsinterpolateex1d REAL(KIND=8) DIMENSION(*) for dfdinterpolate1d/ dfdinterpolateex1d C: float* for dfsInterpolate1D/ dfsInterpolateEx1D double* for dfdInterpolate1D/ dfdInterpolateEx1D	Array for results. If you do not need spline-based interpolation or integration, set this pointer to NULL.
<i>rhint</i>	Fortran: INTEGER C: const MKL_INT	A flag describing the structure of the results. For the list of possible values of <i>rhint</i> , see table "Hint Values for the <i>rhint</i> Parameter". If you set the flag to DF_NO_HINT, the library stores the result in row-major format.
<i>cell</i>	Fortran: INTEGER DIMENSION(*) C: MKL_INT*	Array of cell indices in partition <i>x</i> that contain the interpolation sites. Provide this parameter as input if <i>type</i> is DF_INTERP_USER_CELL. If you do not need cell indices, set this parameter to NULL.
<i>le_cb</i>	Fortran: INTEGER	User-defined callback function for extrapolation at the sites to the left of the interpolation interval.

Name	Type	Description
	C: const dfsInterpCallBack for dfsInterpolateEx1D	Set to NULL if you are not supplying a callback function.
	const dfdInterpCallBack for dfdInterpolateEx1D	
le_params	Fortran: INTEGER DIMENSION(*) C: const void*	Pointer to additional user-defined parameters passed by the library to the <i>le_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.
re_cb	Fortran: INTEGER C: const dfsInterpCallBack for dfsInterpolateEx1D	User-defined callback function for extrapolation at the sites to the right of the interpolation interval. Set to NULL if you are not supplying a callback function.
	const dfdInterpCallBack for dfdInterpolateEx1D	
re_params	Fortran: INTEGER DIMENSION(*) C: const void*	Pointer to additional user-defined parameters passed by the library to the <i>re_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.
i_cb	Fortran: INTEGER C: const dfsInterpCallBack for dfsInterpolateEx1D	User-defined callback function for interpolation within the interpolation interval. Set to NULL if you are not supplying a callback function.
	const dfdInterpCallBack for dfdInterpolateEx1D	
i_params	Fortran: INTEGER DIMENSION(*) C: const void*	Pointer to additional user-defined parameters passed by the library to the <i>i_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.
search_cb	Fortran: INTEGER C: const dfsSearchCellsCallBack for dfdInterpolateEx1D	User-defined callback function for computing indices of cells that can contain interpolation sites. Set to NULL if you are not supplying a callback function.
	const dfdSearchCellsCallBack for dfdInterpolateEx1D	
search_params	Fortran: INTEGER DIMENSION(*) C: const void*	Pointer to additional user-defined parameters passed by the library to the <i>search_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Status of the routine: <ul style="list-style-type: none"> • DF_STATUS_OK if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.
<i>r</i>		Contains results of computations at the interpolation sites.
<i>cell</i>		Array of cell indices in partition <i>x</i> that contain the interpolation sites, which is computed if <i>type</i> is DF_CELL.

Description

The `df?interpolate1d/df?interpolateex1d` routine performs spline-based computations with user-defined settings. The routine supports two types of computations for interpolation sites provided in array *site*:

Computation Types Supported by the `df?interpolate1d/df?interpolateex1d` Routines

Type	Description
DF_INTERP	Compute derivatives of predefined order. The derivative of the zero order is the spline value.
DF_INTERP_USER_CELL	Compute derivatives of predefined order given user-provided cell indices. The derivative of the zero order is the spline value.
	For this type of the computations you should provide a valid <i>cell</i> array, which holds the indices of cells in the <i>site</i> array containing relevant interpolation sites.
DF_CELL	Compute indices of cells in partition <i>x</i> that contain the sites.

If the indices of cells which contain interpolation types are available before the call to `df?interpolate1d/df?interpolateex1d`, you can improve performance by using the DF_INTERP_USER_CELL computation type.

NOTE

If you pass any combination of DF_INTERP, DF_INTERP_USER_CELL, and DF_CELL computation types to the routine, the library uses the DF_INTERP_USER_CELL computation mode.

If you specify DF_INTERP_USER_CELL computation mode and a user-defined callback function for computing cell indices to `df?interpolateex1d`, the library uses the DF_INTERP_USER_CELL computation mode, and the call-back function is not called.

If the sites do not belong to interpolation interval $[a, b]$, the library uses:

- polynomial P_0 of the spline constructed on interval $[x_0, x_1]$ for computations at the sites to the left of a .
- polynomial P_{n-2} of the spline constructed on interval $[x_{n-2}, x_{n-1}]$ for computations at the sites to the right of b .

Interpolation sites support the following hints:

Hint Values for Interpolation Sites

Value	Description
DF_NON_UNIFORM_PARTITION	Partition is non-uniform.
DF_UNIFORM_PARTITION	Partition is uniform.
DF_SORTED_DATA	Interpolation sites are sorted in the ascending order and define a non-uniform partition.
DF_NO_HINT	No hint is provided. By default, the partition defined by interpolation sites is interpreted as non-uniform.

NOTE

If you pass a sorted array of interpolation sites to the Intel MKL, set the *sitehint* parameter to the DF_SORTED_DATA value. The library uses this information when choosing the search algorithm and ignores any other data hints about the structure of the interpolation sites.

Data Fitting computation routines can use the following hints to speed up the computation:

- DF_UNIFORM_PARTITION describes the structure of breakpoints and the interpolation sites.
- DF_QUASI_UNIFORM_PARTITION describes the structure of breakpoints.

Pass the above hints to the library when appropriate.

The *r* pointer defines the memory location for the sets of interpolation and integration results for all coordinates of function *y*. The sets are stored one by one, in the successive order of the function coordinates from *y₁* to *y_n*.

You can define the following settings for packing the results within each set:

- Computation type: interpolation, integration, or both.
- Computation parameters: derivative orders.
- Storage format for the results. You can specify the format using the *rhint* parameter values described in the table below:

Hint Values for the *rhint* Parameter

Value	Description
DF_MATRIX_STORAGE_ROWS	Data is stored in row-major format according to C conventions.
DF_MATRIX_STORAGE_COLS	Data is stored in column-major format according to Fortran conventions.
DF_NO_HINT	No hint is provided. By default, the results are stored in row-major format.

For spline-based interpolation, you should set the derivatives whose values are required for the computation. You can provide the derivatives by setting the *dorder* array of size *ndorder* as follows:

$$dorder[i] = \begin{cases} 1, & \text{if derivative of the } i\text{-th order is required} \\ 0, & \text{otherwise} \end{cases} \quad i = 0, \dots, ndorder - 1$$

See below a common structure of the storage formats of the interpolation results within each set *r* for computing derivatives of order *i₁, i₂, ..., i_m* at *nsite* interpolation sites. In this description, *j* is the coordinate of the vector-valued function:

- Row-major format

$r_j(i_1, 0)$	$r_j(i_2, 0)$...	$r_j(i_m, 0)$
$r_j(i_1, 1)$	$r_j(i_2, 1)$...	$r_j(i_m, 1)$
...
$r_j(i_1, n_{site} - 1)$	$r_j(i_2, n_{site} - 1)$...	$r_j(i_m, n_{site} - 1)$

- Column-major format

$r_j(i_1, 0)$	$r_j(i_1, 1)$...	$r_j(i_1, n_{site} - 1)$
$r_j(i_2, 0)$	$r_j(i_2, 1)$...	$r_j(i_2, n_{site} - 1)$
...
$r_j(i_m, 0)$	$r_j(i_m, 1)$...	$r_j(i_m, n_{site} - 1)$

To speed up Data Fitting computations, use the `datahint` parameter that provides additional information about the structure of the partition and interpolation sites. This data represents a floating-point or a double array with the following structure:

Structure of the `datahint` Array

Element Number	Description
0	Task dimension
1	Type of additional information
2	Reserved field
3	The total number q of elements containing additional information.
4	Element (1)
...	...
$q+3$	Element (q)

Data Fitting computation functions support the following types of additional information for `datahint[1]`:

Types of Additional Information

Type	Element Number	Parameter
<code>DF_NO_APRIORI_INFO</code>	0	No parameters are provided. Information about the data structure is absent.
<code>DF_APRIORI_MOST_LIKELY_CELL</code>	1	Index of the cell that is likely to contain interpolation sites.

To compute indices of the cells that contain interpolation sites, provide the pointer to the array of size n_{site} for the results. The library supports the following scheme of cell indexing for the given partition $\{x_i\}$, $i=1,\dots,n_x$:

$cell[j] = i$, if $site[j] \in [x_i, x_{i+1})$, $i = 0, \dots, n_x$,

where

- $x_0 = -\infty$
- $x_{n_x+1} = +\infty$

- $j = 0, \dots, nsite-1$

To perform interpolation computations with spline types unsupported in the Data Fitting component, use the extended version of the routine `df?interpolateex1d`. With this routine, you can provide user-defined callback functions for computations within, to the left of, or to the right of interpolation interval $[a, b]$. The callback functions compute indices of the cells that contain the specified interpolation sites or can serve as an approximation for computing the exact indices of such cells.

If you do not pass any function for computations at the sites outside the interval $[a, b]$, the routine uses the default settings.

See Also

[Mathematical Conventions for Data Fitting Functions](#)

[df?interpcallback](#)

[df?searchcellcallback](#)

df?integrate1d/df?integrateex1d

Computes a spline-based integral.

Syntax

Fortran:

```
status = dfsintegrate1d(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint,
rdatahint, r, rhint)

status = dfdintegrate1d(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint,
rdatahint, r, rhint)

status = dfsintegrateex1d(task, method, nlim, llim, llimhint, rlim, rlimhint,
ldatahint, rdatahint, r, rhint, le_cb, le_params, re_cb, re_params, i_cb, i_params,
search_cb, search_params)

status = dfdintegrateex1d(task, method, nlim, llim, llimhint, rlim, rlimhint,
ldatahint, rdatahint, r, rhint, le_cb, le_params, re_cb, re_params, i_cb, i_params,
search_cb, search_params)
```

C:

```
status = dfsIntegrate1D(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint,
rdatahint, r, rhint)

status = dfdIntegrate1D(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint,
rdatahint, r, rhint)

status = dfsIntegrateEx1D(task, method, nlim, llim, llimhint, rlim, rlimhint,
ldatahint, rdatahint, r, rhint, le_cb, le_params, re_cb, re_params, i_cb, i_params,
search_cb, search_params)

status = dfdIntegrateEx1D(task, method, nlim, llim, llimhint, rlim, rlimhint,
ldatahint, rdatahint, r, rhint, le_cb, le_params, re_cb, re_params, i_cb, i_params,
search_cb, search_params)
```

Include Files

- Fortran: `mkl_df.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
<i>task</i>	Fortran: TYPE(DF_TASK) C: DFTaskPtr	Descriptor of the task.
<i>method</i>	Fortran: INTEGER C: const MKL_INT	Integration method. The supported value is DF_METHOD_PP.
<i>nlim</i>	Fortran: INTEGER C: const MKL_INT	Number of pairs of integration limits.
<i>llim</i>	Fortran: REAL(KIND=4) DIMENSION(*limits. for dfsintegrate1d/ dfsintegrateex1d REAL(KIND=8) DIMENSION(*) for dfdintegrate1d/ dfdintegrateex1d C: const float* for dfsIntegrate1D/ dfsIntegrateEx1D const double* for dfdIntegrate1D/ dfdIntegrateEx1D	Array of size <i>nlim</i> that defines the left-side integration limits.
<i>llimhint</i>	Fortran: INTEGER C: const MKL_INT	A flag describing the structure of the left-side integration limits <i>llim</i> . For the list of possible values of <i>llimhint</i> , see table " Hint Values for Integration Limits ". If you set the flag to the DF_NO_HINT value, the library assumes that the left-side integration limits define a non-uniform partition.
<i>rlim</i>	Fortran: REAL(KIND=4) DIMENSION(*limits. for dfsintegrate1d/ dfsintegrateex1d REAL(KIND=8) DIMENSION(*) for dfdintegrate1d/ dfdintegrateex1d C: const float* for dfsIntegrate1D/ dfsIntegrateEx1D const double* for dfdIntegrate1D/ dfdIntegrateEx1D	Array of size <i>nlim</i> that defines the right-side integration limits.

Name	Type	Description
<i>rlimhint</i>	Fortran: INTEGER C: const MKL_INT	A flag describing the structure of the right-side integration limits <i>rlim</i> . For the list of possible values of <i>rlimhint</i> , see table " Hint Values for Integration Limits ". If you set the flag to the DF_NO_HINT value, the library assumes that the right-side integration limits define a non-uniform partition.
<i>ldatahint</i>	Fortran: REAL(KIND=4) DIMENSION(*) for dfsintegrate1d/ dfsintegrateex1d REAL(KIND=8) DIMENSION(*) for dfdintegrate1d/ dfdintegrateex1d C: const float* for dfsIntegrate1D/ dfsIntegrateEx1D const double* for dfdIntegrate1D/ dfdIntegrateEx1D	Array that contains additional information about the structure of partition <i>x</i> and left-side integration limits. For details on the <i>ldatahint</i> array, see table " Structure of the datahint Array " in the description of the df?Intepolate1D function.
<i>rdatahint</i>	Fortran: REAL(KIND=4) DIMENSION(*) for dfsintegrate1d/ dfsintegrateex1d REAL(KIND=8) DIMENSION(*) for dfdintegrate1d/ dfdintegrateex1d C: const float* for dfsIntegrate1D/ dfsIntegrateEx1D const double* for dfdIntegrate1D/ dfdIntegrateEx1D	Array that contains additional information about the structure of partition <i>x</i> and right-side integration limits. For details on the <i>rdatahint</i> array, see table " Structure of the datahint Array " in the description of the df?Intepolate1D function.
<i>rhint</i>	Fortran: INTEGER C: const MKL_INT	A flag describing the structure of the results. For the list of possible values of <i>rhint</i> , see table " Hint Values for Integration Results ". If you set the flag to the DF_NO_HINT value, the library stores the results in row-major format.
<i>le_cb</i>	Fortran: INTEGER C: const dfsIntegrCallBack for dfsIntegrateEx1D const dfdIntegrCallBack for dfdIntegrateEx1D	User-defined callback function for integration on interval [<i>llim</i> [<i>i</i>], min(<i>rlim</i> [<i>i</i>], <i>a</i>)) for <i>llim</i> [<i>i</i>] < <i>a</i> . Set to NULL if you are not supplying a callback function.
<i>le_params</i>	Fortran: INTEGER DIMENSION(*)	Pointer to additional user-defined parameters passed by the library to the <i>le_cb</i> function.

Name	Type	Description
	C: const void*	Set to NULL if there are no additional parameters or if you are not supplying a callback function.
<i>re_cb</i>	Fortran: INTEGER C: const dfsInterpCallBack for dfsIntegrateEx1D const dfdInterpCallBack for dfdIntegrateEx1D	User-defined callback function for integration on interval $[\max(l1im[i], b), rlim[i]]$ for $rlim[i] \geq b$. Set to NULL if you are not supplying a callback function.
<i>re_params</i>	Fortran: INTEGER DIMENSION(*) C: const void*	Pointer to additional user-defined parameters passed by the library to the <i>re_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.
<i>i_cb</i>	Fortran: INTEGER C: const dfsIntegrCallBack for dfsIntegrateEx1D const dfdIntegrCallBack for dfdIntegrateEx1D	User-defined callback function for integration on interval $[\max(a, l1im[i],), \min(rlim[i], b))$. Set to NULL if you are not supplying a callback function.
<i>i_params</i>	Fortran: INTEGER DIMENSION(*) C: const void*	Pointer to additional user-defined parameters passed by the library to the <i>i_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.
<i>search_cb</i>	Fortran: INTEGER C: const dfsSearchCellsCallBack for dfsIntegrateEx1D const dfdSearchCellsCallBack for dfdIntegrateEx1D	User-defined callback function for computing indices of cells that can contain interpolation sites. Set to NULL if you are not supplying a callback function.
<i>search_params</i>	Fortran: INTEGER DIMENSION(*) C: const void*	Pointer to additional user-defined parameters passed by the library to the <i>search_cb</i> function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Status of the routine: <ul style="list-style-type: none">• DF_STATUS_OK if the routine execution completed successfully.

Name	Type	Description
<i>r</i>	<p>Fortran:</p> <pre>REAL(KIND=4) DIMENSION(*) for dfsintegrate1d/ dfsintegrateex1d</pre> <p>REAL(KIND=8) DIMENSION(*) for dfdintegrate1d/ dfdintegrateex1d</p> <p>C: float* for dfsIntegrate1D/ dfsIntegrateEx1D</p> <p>double* for dfdIntegrate1D/ dfdIntegrateEx1D</p>	<ul style="list-style-type: none"> Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions. <p>Array of integration results. The size of the array should be sufficient to hold $nlim*ny$ values, where ny is the dimension of the vector-valued function. The integration results are packed according to the settings in <i>rhint</i>.</p>

Description

The `df?integrate1d/df?integrateex1d` routine computes spline-based integral on user-defined intervals

$$I(i, j) = \int_{rlim[i]}^{rlim[i]} f_j(x) dx, i = 0, \dots, nlim - 1, j = 0, \dots, ny - 1$$

If $rlim[i] < llim[i]$, the routine returns

$$I(i, j) = - \int_{rlim[i]}^{llim[i]} f_j(x) dx$$

The routine supports the following hint values for integration results:

Hint Values for Integration Results

Value	Description
<code>DF_MATRIX_STORAGE_ROWS</code>	Data is stored in row-major format according to C conventions.
<code>DF_MATRIX_STORAGE_COLS</code>	Data is stored in column-major format according to Fortran conventions.
<code>DF_NO_HINT</code>	No hint is provided. By default, the coordinates of vector-valued function <i>y</i> are provided and stored in row-major format.

A common structure of the storage formats for the integration results is as follows:

- Row-major format

$I(0, 0)$...	$I(0, nlim - 1)$
...
$I(ny - 1, 0)$...	$I(ny - 1, nlim - 1)$

- Column-major format

$I(0, 0)$...	$I(ny - 1, 0)$
...
$I(0, nlim - 1)$...	$I(ny - 1, nlim - 1)$

Using the `llimhint` and `rlimhint` parameters, you can provide the following hint values for integration limits:

Hint Values for Integration Limits

Value	Description
<code>DF_SORTED_DATA</code>	Integration limits are sorted in the ascending order and define a non-uniform partition.
<code>DF_NON_UNIFORM_PARTITION</code>	Partition defined by integration limits is non-uniform.
<code>DF_UNIFORM_PARTITION</code>	Partition defined by integration limits is uniform.
<code>DF_NO_HINT</code>	No hint is provided. By default, partition defined by integration limits is interpreted as non-uniform.

To compute integration with splines unsupported in the Data Fitting component, use the extended version of the routine `df?integrateex1d`. With this routine, you can provide user-defined callback functions that compute:

- integrals within, to the left of, or to the right of the interpolation interval $[a, b]$
- indices of cells that contain the provided integration limits or can serve as an approximation for computing the exact indices of such cells

If you do not pass callback functions, the routine uses the default settings.

See Also

[Mathematical Conventions for Data Fitting Functions](#)

[df?interpolate1d/df?interpolateex1d](#)

[df?integrcallback](#)

[df?searchcellscallback](#)

df?searchcells1d/df?searchcellsex1d

Searches sub-intervals containing interpolation sites.

Syntax

Fortran:

```
status = dfssearchcells1d(task, method, nsite, site, sitehint, datahint, cell)
status = dfdsearchcells1d(task, method, nsite, site, sitehint, datahint, cell)
status = dfssearchcellsex1d(task, method, nsite, site, sitehint, datahint, cell,
                           search_cb, search_params)
status = dfdsearchcellsex1d(task, method, nsite, site, sitehint, datahint, cell,
                           search_cb, search_params)
```

C:

```
status = dfsSearchCells1D(task, method, nsite, site, sitehint, datahint, cell)
status = dfdSearchCells1D(task, method, nsite, site, sitehint, datahint, cell)
```

```
status = dfsSearchCellsEx1D(task, method, nsite, site, sitehint, datahint, cell,
search_cb, search_params)

status = dfdSearchCellsEx1D(task, method, nsite, site, sitehint, datahint, cell,
search_cb, search_params)
```

Include Files

- Fortran: mkl_df.f90
- C: mkl.h

Input Parameters

Name	Type	Description
task	Fortran: TYPE(DF_TASK) C: DFTaskPtr	Descriptor of the task.
method	Fortran: INTEGER C: const MKL_INT	Search method. The supported value is DF_METHOD_STD.
nsite	Fortran: INTEGER C: const MKL_INT*	Number of interpolation sites.
site	Fortran: REAL(KIND=4) DIMENSION(*) for dfssearchcells1d/ dfssearchcellsex1d REAL(KIND=8) DIMENSION(*) for dfdsearchcells1d/ dfdsearchcellsex1d C: const float* for dfsSearchCells1D/ dfsSearchCellsEx1D const double* for dfdSearchCells1D/ dfdSearchCellsEx1D	Array of interpolation sites of size <i>nsite</i> . The structure of the array is defined by the <i>sitehint</i> parameter: <ul style="list-style-type: none"> • If the sites form a non-uniform partition, the array should contain <i>nsite</i> values. • If the sites form a uniform partition, the array should contain two entries that represent the left-most and the right-most interpolation sites. The first entry of the array contains the left-most interpolation point. The second entry of the array contains the right-most interpolation point.
sitehint	Fortran: INTEGER C: const MKL_INT	A flag describing the structure of the interpolation sites. For the list of possible values of <i>sitehint</i> , see table " Hint Values for Interpolation Sites ". If you set the flag to DF_NO_HINT, the library interprets the site-defined partition as non-uniform.
datahint	Fortran: REAL(KIND=4) DIMENSION(*) for dfssearchcells1d/ dfssearchcellsex1d REAL(KIND=8) DIMENSION(*) for dfdsearchcells1d/ dfdsearchcellsex1d	Array that contains additional information about the structure of the partition and interpolation sites. This data helps to speed up the computation. If you provide a NULL pointer, the routine uses the default settings for computations. For details on the <i>datahint</i> array, see table " Structure of the datahint Array ".

Name	Type	Description
	C: const float* for dfsSearchCells1D/ dfsSearchCellsEx1D const double* for dfdSearchCells1D/ dfdSearchCellsEx1D	
<i>search_cb</i>	Fortran: INTEGER C: const dfsSearchCellsCallBack for dfsSearchCellsEx1D const dfdSearchCellsCallBack for dfdSearchCellsEx1D	User-defined callback function for computing indices of cells that can contain interpolation sites. Set to <code>NULL</code> if you are not supplying a callback function.
<i>search_params</i>	Fortran: INTEGER DIMENSION (*) C: const void*	Pointer to additional user-defined parameters passed by the library to the <i>search_cb</i> function. Set to <code>NULL</code> if there are no additional parameters or if you are not supplying a callback function.

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	Status of the routine: <ul style="list-style-type: none"> • <code>DF_STATUS_OK</code> if the routine execution completed successfully. • Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.
<i>cell</i>	Fortran: INTEGER DIMENSION (*) C: MKL_INT*	Array of cell indices in the partition that contain the interpolation sites.

Description

The `df?searchcells1d/df?searchcellsex1d` routines return array *cell* of indices of sub-intervals (cells) in the partition that contain interpolation sites available in array *site*. For details on the cell indexing scheme, see the description of the `df?interpolate1d/df?interpolateex1d` computation routines.

Use the *datahint* parameter to provide additional information about the structure of the partition and/or interpolation sites. The definition of the *datahint* parameter is available in the description of the `df?interpolate1d/df?interpolateex1d` computation routines.

For description of the user-defined callback for computation of cell indices, see `df?searchcellscallback`.

See Also

[Mathematical Conventions for Data Fitting Functions](#)

[df?interpolate1d/df?interpolateex1d](#)

[df?searchcellscallback](#)

df?interpcallback

A callback function for user-defined interpolation to be passed into df?interpolateex1d.

Syntax

Fortran:

```
status = dfsinterpcallback(n, cell, site, r, params)
status = dfdinterpcallback(n, cell, site, r, params)
```

C:

```
status = dfsInterpCallBack(n, cell, site, r, params)
status = dfdInterpCallBack(n, cell, site, r, params)
```

Include Files

- Fortran: mkl_df.f90
- C: mkl.h

Input Parameters

Name	Type	Description
n	Fortran: INTEGER(KIND=8) C: long long*	Number of interpolation sites.
cell	Fortran: INTEGER(KIND=8) DIMENSION(*) for dfsinterpcallback C: long long*	Array of size n containing indices of the cells to which the interpolation sites in array site belong.
site	Fortran: REAL(KIND=4) DIMENSION(*) for dfsinterpcallback REAL(KIND=8) DIMENSION(*) for dfdinterpcallback C: float* for dfsInterpCallBack double* for dfdInterpCallBack	Array of interpolation sites of size n.
params	Fortran: INTEGER DIMENSION(*) C: void*	Pointer to user-defined parameters of the callback function.

Output Parameters

Name	Type	Description
status	Fortran: INTEGER	The status returned by the callback function:

Name	Type	Description
	C: int	<ul style="list-style-type: none"> • Zero indicates successful completion of the callback operation. • A negative value indicates an error. • A positive value indicates a warning. <p>See "Task Status and Error Reporting" for error code definitions.</p>
r	Fortran: <pre>REAL(KIND=4) DIMENSION(*) for dfsinterpcallback REAL(KIND=8) DIMENSION(*) for dfdinterpcallback C: float* for dfsInterpCallBack double* for dfdInterpCallBack</pre>	Array of the computed interpolation results packed in row-major format.

Description

When passed into the `df?interpolateex1d` routine, this function performs user-defined interpolation operation.

See Also

[df?interpolate1d/df?interpolateex1d](#)
[df?searchcellscallback](#)

df?integrcallback

A callback function that you can pass into `df?integrateex1d` to define integration computations.

Syntax

Fortran:

```
status = dfsintegrcallback(n, lcell, llim, rcell, rlim, r, params)
status = dfdintegrcallback(n, lcell, llim, rcell, rlim, r, params)
```

C:

```
status = dfsIntegrCallBack(n, lcell, llim, rcell, rlim, r, params)
status = dfdIntegrCallBack(n, lcell, llim, rcell, rlim, r, params)
```

Include Files

- **Fortran:** `mkl_df.f90`
- **C:** `mkl.h`

Input Parameters

Name	Type	Description
<i>n</i>	Fortran: INTEGER(KIND=8) C: long long*	Number of pairs of integration limits.
<i>lcell</i>	Fortran: INTEGER(KIND=8) DIMENSION(*) C: long long*	Array of size <i>n</i> with indices of the cells that contain the left-side integration limits in array <i>llim</i> .
<i>llim</i>	Fortran: REAL(KIND=4) DIMENSION(*) for dfsintegrcallback REAL(KIND=8) DIMENSION(*) for dfdintegrcallback C: float* for dfsIntegrCallBack double* for dfdIntegrCallBack	Array of size <i>n</i> that holds the left-side integration limits.
<i>rcell</i>	Fortran: INTEGER(KIND=8) DIMENSION(*) C: long long*	Array of size <i>n</i> with indices of the cells that contain the right-side integration limits in array <i>rlim</i> .
<i>rlim</i>	Fortran: REAL(KIND=4) DIMENSION(*) for dfsintegrcallback REAL(KIND=8) DIMENSION(*) for dfdintegrcallback C: float* for dfsIntegrCallBack double* for dfdIntegrCallBack	Array of size <i>n</i> that holds the right-side integration limits.
<i>params</i>	Fortran: INTEGER DIMENSION(*) C: void*	Pointer to user-defined parameters of the callback function.

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	The status returned by the callback function: <ul style="list-style-type: none"> Zero indicates successful completion of the callback operation. A negative value indicates an error. A positive value indicates a warning.

Name	Type	Description
<i>r</i>	<p>Fortran:</p> <pre>REAL(KIND=4) DIMENSION(*) for dfsintegrcallback</pre> <p>REAL(KIND=8) DIMENSION(*) for dfdintegrcallback</p> <p>C: float* for dfsIntegrCallBack</p> <p>double* for dfdIntegrCallBack</p>	<p>See "Task Status and Error Reporting" for error code definitions.</p> <p>Array of integration results. For packing the results in row-major format, follow the instructions described in df?interpolate1d/df?interpolateex1d.</p>

Description

When passed into the `df?integrateex1d` routine, this function defines integration computations. If at least one of the integration limits is outside the interpolation interval $[a, b]$, the library decomposes the integration into sub-intervals that belong to the extrapolation range to the left of a , the extrapolation range to the right of b , and the interpolation interval $[a, b]$, as follows:

- If the left integration limit is to the left of the interpolation interval ($llim < a$), the `df?integrateex1d` routine passes $llim$ as the left integration limit and $\min(rlim, a)$ as the right integration limit to the user-defined callback function.
- If the right integration limit is to the right of the interpolation interval ($rlim > b$), the `df?integrateex1d` routine passes $\max(llim, b)$ as the left integration limit and $rlim$ as the right integration limit to the user-defined callback function.
- If the left and the right integration limits belong to the interpolation interval, the `df?integrateex1d` routine passes them to the user-defined callback function unchanged.

The value of the integral is the sum of integral values obtained on the sub-intervals.

See Also

[df?integrate1d/df?integrateex1d](#)
[df?integrcallback](#)
[df?searchcellscallback](#)

df?searchcellscallback

A callback function for user-defined search to be passed into `df?interpolateex1d` or `df?searchcellsex1d`.

Syntax

Fortran:

```
status = dfssearchcellscallback(n, site, cell, flag, params)
status = dfdsearchcellscallback(n, site, cell, flag, params)
```

C:

```
status = dfsSearchCellsCallBack(n, site, cell, flag, params)
status = dfdSearchCellsCallBack(n, site, cell, flag, params)
```

Include Files

- Fortran: mkl_df.f90
- C: mkl.h

Input Parameters

Name	Type	Description
<i>n</i>	Fortran: INTEGER(KIND=8) C: long long*	Number of interpolation sites.
<i>site</i>	Fortran: REAL(KIND=4) DIMENSION(*) for dfssearchcellcallback REAL(KIND=8) DIMENSION(*) for dfdsearchcellcallback C: float* for dfsSearchCellsCallBack double* for dfdSearchCellsCallBack	Array of interpolation sites of size <i>n</i> .
<i>flag</i>	Fortran: INTEGER(KIND=4) DIMENSION(*) C: int*	Array of size <i>n</i> , with values set as follows: <ul style="list-style-type: none"> • If the cell with index <i>cell</i>[<i>i</i>] contains <i>site</i>[<i>i</i>], set <i>flag</i>[<i>i</i>] to 1. • Otherwise, set <i>flag</i>[<i>i</i>] to zero. In this case, the library interprets the index as an approximation and computes the index of the cell containing <i>site</i>[<i>i</i>] by using the provided index as a starting point for the search.
<i>params</i>	Fortran: INTEGER DIMENSION(*) C: void*	Pointer to user-defined parameters of the callback function.

Output Parameters

Name	Type	Description
<i>status</i>	Fortran: INTEGER C: int	The status returned by the callback function: <ul style="list-style-type: none"> • Zero indicates successful completion of the callback operation. • A negative value indicates an error. • The DF_STATUS_EXACT_RESULT status indicates that cell indices returned by the callback function are exact. In this case, you do not need to initialize entries of the <i>flag</i> array. • A positive value indicates a warning.
		See " Task Status and Error Reporting " for error code definitions.

Name	Type	Description
cell	Fortran: INTEGER(KIND=8) DIMENSION(*) C: long long*	Array of size n that returns indices of the cells computed by the callback function.

Description

When passed into the `df?interpolateex1d` or `df?searchcellsex1d` routine, this function performs a user-defined search.

See Also

`df?interpolate1d/df?interpolateex1d`
`df?interpcallback`

Task Destructors

Task destructors are routines used to delete task descriptors and deallocate the corresponding memory resources. The Data Fitting task destructor `dfdeletetask` destroys a Data Fitting task and frees the memory.

dfdeletetask

Destroys a Data Fitting task object and frees the memory.

Syntax

Fortran:

```
status = dfdeletetask(task)
```

C:

```
status = dfDeleteTask(&task)
```

Include Files

- Fortran: `mkl_df.f90`
- C: `mkl.h`

Input Parameters

Name	Type	Description
task	Fortran: <code>TYPE(DF_TASK)</code> C: <code>DFTaskPtr</code>	Descriptor of the task to destroy.

Output Parameters

Name	Type	Description
status	Fortran: <code>INTEGER</code> C: <code>int</code>	Status of the routine: <ul style="list-style-type: none"> <code>DF_STATUS_OK</code> if the task is deleted successfully. Non-zero error code if the operation failed. See "Task Status and Error Reporting" for error code definitions.

Description

Given a pointer to a task descriptor, this routine deletes the Data Fitting task descriptor and frees the memory allocated for the structure. If the task is deleted successfully, the routine sets the task pointer to NULL. Otherwise, the routine returns an error code.

A

Linear Solvers Basics

Many applications in science and engineering require the solution of a system of linear equations. This problem is usually expressed mathematically by the matrix-vector equation, $Ax = b$, where A is an m -by- n matrix, x is the n element column vector and b is the m element column vector. The matrix A is usually referred to as the coefficient matrix, and the vectors x and b are referred to as the solution vector and the right-hand side, respectively.

Basic concepts related to solving linear systems with sparse matrices are described in section [Sparse Linear Systems](#) that follows.

Sparse Linear Systems

In many real-life applications, most of the elements in A are zero. Such a matrix is referred to as sparse. Conversely, matrices with very few zero elements are called dense. For sparse matrices, computing the solution to the equation $Ax = b$ can be made much more efficient with respect to both storage and computation time, if the sparsity of the matrix can be exploited. The more an algorithm can exploit the sparsity without sacrificing the correctness, the better the algorithm.

Generally speaking, computer software that finds solutions to systems of linear equations is called a solver. A solver designed to work specifically on sparse systems of equations is called a sparse solver. Solvers are usually classified into two groups - direct and iterative.

Iterative Solvers start with an initial approximation to a solution and attempt to estimate the difference between the approximation and the true result. Based on the difference, an iterative solver calculates a new approximation that is closer to the true result than the initial approximation. This process is repeated until the difference between the approximation and the true result is sufficiently small. The main drawback to iterative solvers is that the rate of convergence depends greatly on the values in the matrix A . Consequently, it is not possible to predict how long it will take for an iterative solver to produce a solution. In fact, for ill-conditioned matrices, the iterative process will not converge to a solution at all. However, for well-conditioned matrices it is possible for iterative solvers to converge to a solution very quickly. Consequently for the right applications, iterative solvers can be very efficient.

Direct Solvers, on the other hand, often factor the matrix A into the product of two triangular matrices and then perform a forward and backward triangular solve.

This approach makes the time required to solve a systems of linear equations relatively predictable, based on the size of the matrix. In fact, for sparse matrices, the solution time can be predicted based on the number of non-zero elements in the array A .

Matrix Fundamentals

A matrix is a rectangular array of either real or complex numbers. A matrix is denoted by a capital letter; its elements are denoted by the same lower case letter with row/column subscripts. Thus, the value of the element in row i and column j in matrix A is denoted by $a(i, j)$. For example, a 3 by 4 matrix A , is written as follows:

$$A = \begin{bmatrix} a(1, 1) & a(1, 2) & a(1, 3) & a(1, 4) \\ a(2, 1) & a(2, 2) & a(2, 3) & a(2, 4) \\ a(3, 1) & a(3, 2) & a(3, 3) & a(3, 4) \end{bmatrix}$$

Note that with the above notation, we assume the standard Fortran programming language convention of starting array indices at 1 rather than the C programming language convention of starting them at 0.

A matrix in which all of the elements are real numbers is called a real matrix. A matrix that contains at least one complex number is called a complex matrix. A real or complex matrix A with the property that $a(i,j) = a(j,i)$, is called a symmetric matrix. A complex matrix A with the property that $a(i,j) = \text{conj}(a(j,i))$, is called a Hermitian matrix. Note that programs that manipulate symmetric and Hermitian matrices need only store half of the matrix values, since the values of the non-stored elements can be quickly reconstructed from the stored values.

A matrix that has the same number of rows as it has columns is referred to as a square matrix. The elements in a square matrix that have same row index and column index are called the diagonal elements of the matrix, or simply the diagonal of the matrix.

The transpose of a matrix A is the matrix obtained by “flipping” the elements of the array about its diagonal. That is, we exchange the elements $a(i,j)$ and $a(j,i)$. For a complex matrix, if we both flip the elements about the diagonal and then take the complex conjugate of the element, the resulting matrix is called the Hermitian transpose or conjugate transpose of the original matrix. The transpose and Hermitian transpose of a matrix A are denoted by A^T and A^H respectively.

A column vector, or simply a vector, is a $n \times 1$ matrix, and a row vector is a $1 \times n$ matrix. A real or complex matrix A is said to be positive definite if the vector-matrix product $x^T Ax$ is greater than zero for all non-zero vectors x . A matrix that is not positive definite is referred to as indefinite.

An upper (or lower) triangular matrix, is a square matrix in which all elements below (or above) the diagonal are zero. A unit triangular matrix is an upper or lower triangular matrix with all 1's along the diagonal.

A matrix P is called a permutation matrix if, for any matrix A , the result of the matrix product PA is identical to A except for interchanging the rows of A . For a square matrix, it can be shown that if PA is a permutation of the rows of A , then AP^T is the same permutation of the columns of A . Additionally, it can be shown that the inverse of P is P^T .

In order to save space, a permutation matrix is usually stored as a linear array, called a permutation vector, rather than as an array. Specifically, if the permutation matrix maps the i -th row of a matrix to the j -th row, then the i -th element of the permutation vector is j .

A matrix with non-zero elements only on the diagonal is called a diagonal matrix. As is the case with a permutation matrix, it is usually stored as a vector of values, rather than as a matrix.

Direct Method

For solvers that use the direct method, the basic technique employed in finding the solution of the system $Ax = b$ is to first factor A into triangular matrices. That is, find a lower triangular matrix L and an upper triangular matrix U , such that $A = LU$. Having obtained such a factorization (usually referred to as an LU decomposition or LU factorization), the solution to the original problem can be rewritten as follows.

$$\begin{aligned} Ax &= b \\ \Rightarrow LUX &= b \\ \Rightarrow L(Ux) &= b \end{aligned}$$

This leads to the following two-step process for finding the solution to the original system of equations:

1. Solve the systems of equations $Ly = b$.
2. Solve the system $Ux = y$.

Solving the systems $Ly = b$ and $Ux = y$ is referred to as a forward solve and a backward solve, respectively.

If a symmetric matrix A is also positive definite, it can be shown that A can be factored as LL^T where L is a lower triangular matrix. Similarly, a Hermitian matrix, A , that is positive definite can be factored as $A = LL^H$. For both symmetric and Hermitian matrices, a factorization of this form is called a Cholesky factorization.

In a Cholesky factorization, the matrix U in an LU decomposition is either L^T or L^H . Consequently, a solver can increase its efficiency by only storing L , and one-half of A , and not computing U . Therefore, users who can express their application as the solution of a system of positive definite equations will gain a significant performance improvement over using a general representation.

For matrices that are symmetric (or Hermitian) but not positive definite, there are still some significant efficiencies to be had. It can be shown that if A is symmetric but not positive definite, then A can be factored as $A = LDL^T$, where D is a diagonal matrix and L is a lower unit triangular matrix. Similarly, if A is Hermitian, it can be factored as $A = LDL^H$. In either case, we again only need to store L , D , and half of A and we need not compute U . However, the backward solve phases must be amended to solving $L^T x = D^{-1} y$ rather than $L^T x = y$.

Fill-In and Reordering of Sparse Matrices

Two important concepts associated with the solution of sparse systems of equations are fill-in and reordering. The following example illustrates these concepts.

Consider the system of linear equation $Ax = b$, where A is a symmetric positive definite sparse matrix, and A and b are defined by the following:

$$A = \begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ \frac{3}{2} & 1 & * & * & * \\ 6 & * & 12 & * & * \\ \frac{3}{4} & * & * & 5 & * \\ 3 & * & * & 8 & 16 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

A star (*) is used to represent zeros and to emphasize the sparsity of A . The Cholesky factorization of A is: $A = LL^T$, where L is the following:

$$\dots \begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{matrix} \dots$$

Notice that even though the matrix A is relatively sparse, the lower triangular matrix L has no zeros below the diagonal. If we computed L and then used it for the forward and backward solve phase, we would do as much computation as if A had been dense.

The situation of L having non-zeros in places where A has zeros is referred to as fill-in. Computationally, it would be more efficient if a solver could exploit the non-zero structure of A in such a way as to reduce the fill-in when computing L . By doing this, the solver would only need to compute the non-zero entries in L . Toward this end, consider permuting the rows and columns of A . As described in [Matrix Fundamentals](#) section, the permutations of the rows of A can be represented as a permutation matrix, P . The result of permuting the rows is the product of P and A . Suppose, in the above example, we swap the first and fifth row

of A , then swap the first and fifth columns of A , and call the resulting matrix B . Mathematically, we can express the process of permuting the rows and columns of A to get B as $B = PAP^T$. After permuting the rows and columns of A , we see that B is given by the following:

$$B = \begin{bmatrix} 16 & * & * & * & 3 \\ * & \frac{1}{2} & * & * & \frac{3}{2} \\ * & * & 12 & * & 6 \\ * & * & * & \frac{5}{8} & \frac{3}{4} \\ 3 & \frac{3}{2} & 6 & \frac{3}{4} & 9 \end{bmatrix}$$

Since B is obtained from A by simply switching rows and columns, the numbers of non-zero entries in A and B are the same. However, when we find the Cholesky factorization, $B = LL^T$, we see the following:

$$L = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{3}}{4} \end{bmatrix}$$

The fill-in associated with B is much smaller than the fill-in associated with A . Consequently, the storage and computation time needed to factor B is much smaller than to factor A . Based on this, we see that an efficient sparse solver needs to find permutation P of the matrix A , which minimizes the fill-in for factoring $B = PAP^T$, and then use the factorization of B to solve the original system of equations.

Although the above example is based on a symmetric positive definite matrix and a Cholesky decomposition, the same approach works for a general LU decomposition. Specifically, let P be a permutation matrix, $B = PAP^T$ and suppose that B can be factored as $B = LU$. Then

$$Ax = b$$

$$\Rightarrow PA(P^{-1}P)x = Pb$$

$$\Rightarrow PA(P^TP)x = Pb$$

$$\Rightarrow (PAP^T)(Px) = Pb$$

$$\Rightarrow B(Px) = Pb$$

$$\Rightarrow LU(Px) = Pb$$

It follows that if we obtain an LU factorization for B , we can solve the original system of equations by a three step process:

1. Solve $Ly = Pb$.
 2. Solve $Uz = y$.
 3. Set $x = P^T z$.

If we apply this three-step process to the current example, we first need to perform the forward solve of the systems of equation $Ly = Pb$:

$$L_Y = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{3}}{4} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 3 \\ 4 \\ 1 \end{bmatrix}$$

$$\text{This gives: } Y^r = \frac{5}{4}, 2\sqrt{2}, \frac{\sqrt{3}}{2}, \frac{16}{\sqrt{10}}, \frac{-979\sqrt{\frac{3}{5}}}{12}.$$

The second step is to perform the backward solve, $Uz = y$. Or, in this case, since a Cholesky factorization is used, $L^T z = y$.

$$\begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{5}}{4} \end{bmatrix}^T = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \end{bmatrix} = \begin{bmatrix} \frac{5}{4} \\ 2(\sqrt{2}) \\ \frac{\sqrt{3}}{2} \\ \frac{16}{\sqrt{10}} \\ -979\sqrt{\frac{3}{5}} \end{bmatrix}$$

This gives $z^T = \frac{123}{2}, 983, \frac{1961}{12}, 398, \frac{-979}{3}$.

The third and final step is to set $x = P^T z$. This gives $X^T = \frac{-979}{3}, 983, \frac{1961}{12}, 398, \frac{123}{2}$.

Sparse Matrix Storage Formats

As discussed above, it is more efficient to store only the non-zero elements of a sparse matrix. There are a number of common storage formats used for sparse matrices, but most of them employ the same basic technique. That is, store all non-zero elements of the matrix into a linear array and provide auxiliary arrays to describe the locations of the non-zero elements in the original matrix.

Storage Formats for the Direct Sparse Solvers

Storing the non-zero elements of a sparse matrix into a linear array is done by walking down each column (column-major format) or across each row (row-major format) in order, and writing the non-zero elements to a linear array in the order they appear in the walk.

For symmetric matrices, it is necessary to store only the upper triangular half of the matrix (upper triangular format) or the lower triangular half of the matrix (lower triangular format).

The Intel MKL direct sparse solvers use a row-major upper triangular storage format: the matrix is compressed row-by-row and for symmetric matrices only non-zero elements in the upper triangular half of the matrix are stored.

The Intel MKL sparse matrix storage format for direct sparse solvers is specified by three arrays: *values*, *columns*, and *rowIndex*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix.

<i>values</i>	A real or complex array that contains the non-zero elements of a sparse matrix. The non-zero elements are mapped into the <i>values</i> array using the row-major upper triangular storage mapping described above.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column that contains the <i>i</i> -th element in the <i>values</i> array.
<i>rowIndex</i>	Element <i>j</i> of the integer array <i>rowIndex</i> gives the index of the element in the <i>values</i> array that is first non-zero element in a row <i>j</i> .

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in the matrix.

As the *rowIndex* array gives the location of the first non-zero element within a row, and the non-zero elements are stored consecutively, the number of non-zero elements in the *i*-th row is equal to the difference of *rowIndex(i)* and *rowIndex(i+1)*.

To have this relationship hold for the last row of the matrix, an additional entry (dummy entry) is added to the end of *rowIndex*. Its value is equal to the number of non-zero elements plus one. This makes the total length of the *rowIndex* array one larger than the number of rows in the matrix.

NOTE

The Intel MKL sparse storage scheme for the direct sparse solvers supports both one-based indexing and zero-based indexing.

Consider the symmetric matrix *A*:

$$A = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -3 & * & 6 & 7 & * \\ * & * & 4 & * & -5 \end{bmatrix}$$

Only elements from the upper triangle are stored. The actual arrays for the matrix A are as follows:

Storage Arrays for a Symmetric Matrix

one-based indexing

<i>values</i>	=	(1	-1	-3	5	4	6	4	7	-5)
<i>columns</i>	=	(1	2	4	2	3	4	5	4	5)
<i>rowIndex</i>	=	(1	4	5	8	9	10)			

zero-based indexing

<i>values</i>	=	(1	-1	-3	5	4	6	4	7	-5)
<i>columns</i>	=	(0	1	3	1	2	3	4	3	4)
<i>rowIndex</i>	=	(0	3	4	7	8	9)			

For a non-symmetric or non-Hermitian matrix, all non-zero elements need to be stored. Consider the non-symmetric matrix B :

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & * & * & -5 \end{bmatrix}$$

The matrix B has 13 non-zero elements, and all of them are stored as follows:

Storage Arrays for a Non-Symmetric Matrix

one-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
<i>rowIndex</i>	=	(1	4	6	9	12	14)							

zero-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
<i>rowIndex</i>	=	(0	3	5	8	11	13)							

Direct sparse solvers can also solve symmetrically structured systems of equations. A symmetrically structured system of equations is one where the pattern of non-zero elements is symmetric. That is, a matrix has a symmetric structure if $a_{j,i}$ is not zero if and only if $a_{i,j}$ is not zero. From the point of view of the solver software, a "non-zero" element of a matrix is any element stored in the *values* array, even if its value is

equal to 0. In that sense, any non-symmetric matrix can be turned into a symmetrically structured matrix by carefully adding zeros to the *values* array. For example, the above matrix *B* can be turned into a symmetrically structured matrix by adding two non-zero entries:

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -2 & 5 & * & * & 0 \\ * & * & 4 & 6 & 4 \\ -4 & * & 2 & 7 & * \\ * & 8 & 0 & * & -5 \end{bmatrix}$$

The matrix *B* can be considered to be symmetrically structured with 15 non-zero elements and represented as:

Storage Arrays for a Symmetrically Structured Matrix

one-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	0	4	6	4	-4	2	7	8	0	-5)
<i>columns</i>	=	(1	2	4	1	2	5	3	4	5	1	3	4	2	3	5)
<i>rowIndex</i>	=	(1	4	7	10	13	16)									

zero-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	0	4	6	4	-4	2	7	8	0	-5)
<i>columns</i>	=	(0	1	3	0	1	4	2	3	4	0	2	3	1	2	4)
<i>rowIndex</i>	=	(0	3	6	9	12	15)									

The distributed assembled matrix input format can be used by the Parallel Direct Sparse Solver for Clusters Interface.

In this format, the symmetric input matrix *A* is divided into sequential row subsets, or domains. Each domain belongs to an MPI process. Neighboring domains can overlap. For such intersections, the element values of the full matrix can be obtained by summing the respective elements of both domains.

As in the centralized format, the distributed format uses three arrays to describe the input data, but the *values*, *columns*, and *rowIndex* arrays on each processor only describe the domain belonging to the that particular processor and not the entire matrix.

For example, consider a symmetric matrix *A*:

$$A = \begin{bmatrix} 6 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 11 & 5 & 4 \\ -3 & * & 5 & 10 & * \\ * & * & 4 & * & 5 \end{bmatrix}$$

This array could be distributed between two domains corresponding to two MPI processes, with the first containing rows 1 through 3, and the second containing rows 3 through 5.

NOTE

For the symmetric input matrix, it is not necessary to store the values from the lower triangle.

$$A_{\text{Domain1}} = \begin{bmatrix} 6 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 3 & * & 2 \end{bmatrix}$$

Distributed Storage Arrays for a Symmetric Matrix, Domain 1**one-based indexing**

<i>values</i>	=	(6	-1	-3	5	3	2)
<i>columns</i>	=	(1	2	4	2	3	5)
<i>rowIndex</i>	=	(1	4	5	7)		

zero-based indexing

<i>values</i>	=	(6	-1	-3	5	3	2)
<i>columns</i>	=	(0	1	3	1	2	4)
<i>rowIndex</i>	=	(0	3	4	6)		

$$A_{\text{Domain2}} = \begin{bmatrix} * & * & 8 & 5 & 2 \\ -3 & * & 5 & 10 & * \\ * & * & 4 & * & 5 \end{bmatrix}$$

Distributed Storage Arrays for a Symmetric Matrix, Domain 2**one-based indexing**

<i>values</i>	=	(8	5	2	10	5)
<i>columns</i>	=	(3	4	5	4	5)
<i>rowIndex</i>	=	(1	4	5	6)	

zero-based indexing

<i>values</i>	=	(8	5	2	10	5)
<i>columns</i>	=	(2	3	4	3	4)
<i>rowIndex</i>	=	(0	3	4	5)	

The third row of matrix A is common between domain 1 and domain 2. The values of row 3 of matrix A are the sums of the respective elements of row 3 of matrix A_{Domain1} and row 1 of matrix A_{Domain2} .

Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the *values* array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the *values* array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the *values* array.

Sparse Matrix Storage Formats for Sparse BLAS Levels 2 and Level 3

This section describes in detail the sparse matrix storage formats supported in the current version of the Intel MKL Sparse BLAS Level 2 and Level 3.

CSR Format

The Intel MKL compressed sparse row (CSR) format is specified by four arrays: the *values*, *columns*, *pointerB*, and *pointerE*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix *A*.

<i>values</i>	A real or complex array that contains the non-zero elements of <i>A</i> . Values of the non-zero elements of <i>A</i> are mapped into the <i>values</i> array using the row-major storage mapping described above.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column in <i>A</i> that contains the <i>i</i> -th value in the <i>values</i> array.
<i>pointerB</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>values</i> array that is first non-zero element in a row <i>j</i> of <i>A</i> . Note that this index is equal to <i>pointerB(j)</i> - <i>pointerB(1)</i> + 1 .
<i>pointerE</i>	An integer array that contains row indices, such that <i>pointerE(j)</i> - <i>pointerB(1)</i> is the index of the element in the <i>values</i> array that is last non-zero element in a row <i>j</i> of <i>A</i> .

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in *A*. The length of the *pointerB* and *pointerE* arrays is equal to the number of rows in *A*.

NOTE

Note that the Intel MKL Sparse BLAS routines support the CSR format both with one-based indexing and zero-based indexing.

The matrix *B*

$$\dots \begin{pmatrix} 1 & 2 & 4 & 1 & 2 & 3 & 4 & 5 & 1 & 3 & 4 & 2 & 7 & 8 & -5 \end{pmatrix} \dots$$

can be represented in the CSR format as:

Storage Arrays for a Matrix in CSR Format

one-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
<i>pointerB</i>	=	(1	4	6	9	12)								
<i>pointerE</i>	=	(4	6	9	12	14)								

zero-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
<i>pointerB</i>	=	(0	3	5	8	11)								
<i>pointerE</i>	=	(3	5	8	11	13)								

This storage format is used in the NIST Sparse BLAS library [Rem05].

Note that the storage format accepted for the direct sparse solvers and described above (see [Storage Formats for the Direct Sparse Solvers](#)) is a variation of the CSR format. It also is used in the Intel MKL Sparse BLAS Level 2 both with one-based indexing and zero-based indexing. The above matrix B can be represented in this format (referred to as the 3-array variation of the CSR format or CSR3) as:

[Storage Arrays for a Matrix in CSR Format \(3-Array Variation\)](#)

one-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(1	2	4	1	2	3	4	5	1	3	4	2	5)
<i>rowIndex</i>	=	(1	4	6	9	12	14)							

zero-based indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>columns</i>	=	(0	1	3	0	1	2	3	4	0	2	3	1	4)
<i>rowIndex</i>	=	(0	3	5	8	11	13)							

The 3-array variation of the CSR format has a restriction: all non-zero elements are stored continuously, that is the set of non-zero elements in the row J goes just after the set of non-zero elements in the row $J-1$.

There are no such restrictions in the general (NIST) CSR format. This may be useful, for example, if there is a need to operate with different submatrices of the matrix at the same time. In this case, it is enough to define the arrays *pointerB* and *pointerE* for each needed submatrix so that all these arrays are pointers to the same array *values*.

Comparing the array *rowIndex* from the [Table "Storage Arrays for a Non-Symmetric Example Matrix"](#) with the arrays *pointerB* and *pointerE* from the [Table "Storage Arrays for an Example Matrix in CSR Format"](#) it is easy to see that

```
pointerB(i) = rowIndex(i) for i=1, ..5;
pointerE(i) = rowIndex(i+1) for i=1, ..5.
```

This enables calling a routine that has *values*, *columns*, *pointerB* and *pointerE* as input parameters for a sparse matrix stored in the format accepted for the direct sparse solvers. For example, a routine with the interface:

```
Subroutine name_routine(.... , values, columns, pointerB, pointerE, ...)
```

can be called with parameters *values*, *columns*, *rowIndex* as follows:

```
call name_routine(.... , values, columns, rowIndex, rowindex(2), ...).
```

[CSC Format](#)

The compressed sparse column format (CSC) is similar to the CSR format, but the columns are used instead the rows. In other words, the CSC format is identical to the CSR format for the transposed matrix. The CSR format is specified by four arrays: *values*, *columns*, *pointerB*, and *pointerE*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix A .

<i>values</i>	A real or complex array that contains the non-zero elements of A . Values of the non-zero elements of A are mapped into the <i>values</i> array using the column-major storage mapping.
<i>rows</i>	Element i of the integer array <i>rows</i> is the number of the row in A that contains the i -th value in the <i>values</i> array.
<i>pointerB</i>	Element j of this integer array gives the index of the element in the <i>values</i> array that is first non-zero element in a column j of A . Note that this index is equal to $pointerB(j) - pointerB(1)+1$.

<i>pointerE</i>	An integer array that contains column indices, such that $\text{pointerE}(j) - \text{pointerB}(1)$ is the index of the element in the <i>values</i> array that is last non-zero element in a column j of A .
-----------------	--

The length of the *values* and *columns* arrays is equal to the number of non-zero elements in A . The length of the *pointerB* and *pointerE* arrays is equal to the number of columns in A .

NOTE

Note that the Intel MKL Sparse BLAS routines support the CSC format both with one-based indexing and zero-based indexing.

The above matrix B can be represented in the CSC format as:

Storage Arrays for a Matrix in CSC Format

one-based indexing

<i>values</i>	=	(1	-2	-4	-1	5	8	4	2	-3	6	7	4	-5)
<i>rows</i>	=	(1	2	4	1	2	5	3	4	1	3	4	2	5)
<i>pointerB</i>	=	(1	4	7	9	12)								
<i>pointerE</i>	=	(4	7	9	12	14)								

zero-based indexing

<i>values</i>	=	(1	-2	-4	-1	5	8	4	2	-3	6	7	4	-5)
<i>rows</i>	=	(0	1	3	0	1	4	2	3	0	2	3	1	4)
<i>pointerB</i>	=	(0	3	6	8	11)								
<i>pointerE</i>	=	(3	6	8	11	13)								

Coordinate Format

The coordinate format is the most flexible and simplest format for the sparse matrix representation. Only non-zero elements are stored, and the coordinates of each non-zero element are given explicitly. Many commercial libraries support the matrix-vector multiplication for the sparse matrices in the coordinate format.

The Intel MKL coordinate format is specified by three arrays: *values*, *rows*, and *column*, and a parameter *nnz* which is number of non-zero elements in A . All three arrays have dimension *nnz*. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix A .

<i>values</i>	A real or complex array that contains the non-zero elements of A in any order.
<i>rows</i>	Element i of the integer array <i>rows</i> is the number of the row in A that contains the i -th value in the <i>values</i> array.
<i>columns</i>	Element i of the integer array <i>columns</i> is the number of the column in A that contains the i -th value in the <i>values</i> array.

NOTE

Note that the Intel MKL Sparse BLAS routines support the coordinate format both with one-based indexing and zero-based indexing.

For example, the sparse matrix C

$$c = \begin{bmatrix} 1 & -1 & -3 & 0 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{bmatrix}$$

can be represented in the coordinate format as follows:

Storage Arrays for an Example Matrix in case of the coordinate format

one-based indexing

$$\begin{array}{lcl} \text{values} & = & (1 \quad -1 \quad -3 \quad -2 \quad 5 \quad 4 \quad 6 \quad 4 \quad -4 \quad 2 \quad 7 \quad 8 \quad -5) \\ \text{rows} & = & (1 \quad 1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 3 \quad 3 \quad 4 \quad 4 \quad 4 \quad 5 \quad 5) \\ \text{columns} & = & (1 \quad 2 \quad 3 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 1 \quad 3 \quad 4 \quad 2 \quad 5) \end{array}$$

zero-based

indexing

<i>values</i>	=	(1	-1	-3	-2	5	4	6	4	-4	2	7	8	-5)
<i>rows</i>	=	(0	0	0	1	1	2	2	2	3	3	3	4	4)
<i>columns</i>	=	(0	1	2	0	1	2	3	4	0	2	3	1	4)

Diagonal Storage Format

If the sparse matrix has diagonals containing only zero elements, then the diagonal storage format can be used to reduce the amount of information needed to locate the non-zero elements. This storage format is particularly useful in many applications where the matrix arises from a finite element or finite difference discretization. The Intel MKL diagonal storage format is specified by two arrays: *values* and *distance*, and two parameters: *ndiag*, which is the number of non-empty diagonals, and *lval*, which is the declared leading dimension in the calling (sub)programs. The following table describes the arrays *values* and *distance*:

values A real or complex two-dimensional array is dimensioned as *lval* by *ndiag*. Each column of it contains the non-zero elements of certain diagonal of *A*. The key point of the storage is that each element in *values* retains the row number of the original matrix. To achieve this diagonals in the lower triangular part of the matrix are padded from the top, and those in the upper triangular part are padded from the bottom. Note that the value of *distance*(*i*) is the number of elements to be padded for diagonal *i*.

distance An integer array with dimension *ndiag*. Element *i* of the array *distance* is the distance between *i*-diagonal and the main diagonal. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

The above matrix C can be represented in the diagonal storage format as follows:

[View all posts by **John Doe**](#) [View all posts in **Category A**](#) [View all posts in **Category B**](#)

1990-1991: *Journal of the American Academy of Child and Adolescent Psychiatry*, 29(12), 1811-1812.

where the asterisks denote padded elements.

When storing symmetric, Hermitian, or skew-symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For the Intel MKL triangular solver routines elements of the array *distance* must be sorted in increasing order. In all other cases the diagonals and distances can be stored in arbitrary order.

Skyline Storage Format

The skyline storage format is important for the direct sparse solvers, and it is well suited for Cholesky or LU decomposition when no pivoting is required.

The skyline storage format accepted in Intel MKL can store only triangular matrix or triangular part of a matrix. This format is specified by two arrays: *values* and *pointers*. The following table describes these arrays:

<i>values</i>	A scalar array. For a lower triangular matrix it contains the set of elements from each row of the matrix starting from the first non-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets.
<i>pointers</i>	An integer array with dimension $(m+1)$, where m is the number of rows for lower triangle (columns for the upper triangle). $\text{pointers}(i) - \text{pointers}(1)+1$ gives the index of element in <i>values</i> that is first non-zero element in row (column) i . The value of $\text{pointers}(m+1)$ is set to $\text{nnz} + \text{pointers}(1)$, where nnz is the number of elements in the array <i>values</i> .

For example, the low triangle of the matrix C given above can be stored as follows:

```
values = ( 1 -2 5 4 -4 0 2 7 8 0 0 -5 )
pointers = ( 1 2 4 5 9 13 )
```

and the upper triangle of this matrix C can be stored as follows:

```
values = ( 1 -1 5 -3 0 4 6 7 4 0 -5 )
pointers = ( 1 2 4 7 9 12 )
```

This storage format is supported by the NIST Sparse BLAS library [Rem05].

Note that the Intel MKL Sparse BLAS routines operating with the skyline storage format do not support general matrices.

BSR Format

The Intel MKL block compressed sparse row (BSR) format for sparse matrices is specified by four arrays: *values*, *columns*, *pointerB*, and *pointerE*. The following table describes these arrays.

<i>values</i>	A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block-by-block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them are equal to zero. Within each non-zero block elements are stored in column-major order in the case of one-based indexing, and in row-major order in the case of the zero-based indexing.
---------------	---

<i>columns</i>	Element i of the integer array <i>columns</i> is the number of the column in the block matrix that contains the i -th non-zero block.
<i>pointerB</i>	Element j of this integer array gives the index of the element in the <i>columns</i> array that is first non-zero block in a row j of the block matrix.
<i>pointerE</i>	Element j of this integer array gives the index of the element in the <i>columns</i> array that contains the last non-zero block in a row j of the block matrix plus 1.

The length of the *values* array is equal to the number of all elements in the non-zero blocks, the length of the *columns* array is equal to the number of non-zero blocks. The length of the *pointerB* and *pointerE* arrays is equal to the number of block rows in the block matrix.

NOTE

Note that the Intel MKL Sparse BLAS routines support BSR format both with one-based indexing and zero-based indexing.

For example, consider the sparse matrix D

$$D = \begin{bmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ * & * & 1 & 4 & * & * \\ * & * & 5 & 1 & * & * \\ * & * & 4 & 3 & 7 & 2 \\ * & * & 0 & 0 & 0 & 0 \end{bmatrix}$$

If the size of the block equals 2, then the sparse matrix D can be represented as a 3×3 block matrix E with the following structure:

$$E = \begin{bmatrix} L & M & * \\ * & N & * \\ * & P & Q \end{bmatrix}$$

where

$$L = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}, M = \begin{bmatrix} 6 & 7 \\ 8 & 2 \end{bmatrix}, N = \begin{bmatrix} 1 & 4 \\ 5 & 1 \end{bmatrix}, P = \begin{bmatrix} 4 & 3 \\ 0 & 0 \end{bmatrix}, Q = \begin{bmatrix} 7 & 2 \\ 0 & 0 \end{bmatrix}$$

The matrix D can be represented in the BSR format as follows:

one-based indexing

```

values = (1 2 0 1 6 8 7 2 1 5 4 1 4 0 3 0 7 0 2 0)
columns = (1 2 2 3)
pointerB = (1 3 4)
pointerE = (3 4 6)

```

zero-based indexing

```

values = [1 0 2 1 6 7 8 2 1 4 5 1 4 3 0 0 7 2 0 0]
columns = [0 1 1 2]
pointerB = [0 2 3]
pointerE = [2 3 5]

```

This storage format is supported by the NIST Sparse BLAS library [[Rem05](#)].

Intel MKL supports the variation of the BSR format that is specified by three arrays: *values*, *columns*, and *rowIndex*. The following table describes these arrays.

<i>values</i>	A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block by block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them is equal to zero. Within each non-zero block the elements are stored in column major order in the case of the one-based indexing, and in row major order in the case of the zero-based indexing.
<i>columns</i>	Element <i>i</i> of the integer array <i>columns</i> is the number of the column in the block matrix that contains the <i>i</i> -th non-zero block.
<i>rowIndex</i>	Element <i>j</i> of this integer array gives the index of the element in the <i>columns</i> array that is first non-zero block in a row <i>j</i> of the block matrix.

The length of the *values* array is equal to the number of all elements in the non-zero blocks, the length of the *columns* array is equal to the number of non-zero blocks.

As the *rowIndex* array gives the location of the first non-zero block within a row, and the non-zero blocks are stored consecutively, the number of non-zero blocks in the *i*-th row is equal to the difference of *rowIndex*(*i*) and *rowIndex*(*i*+1).

To retain this relationship for the last row of the block matrix, an additional entry (dummy entry) is added to the end of *rowIndex* with value equal to the number of non-zero blocks plus one. This makes the total length of the *rowIndex* array one larger than the number of rows of the block matrix.

The above matrix *D* can be represented in this 3-array variation of the BSR format as follows:

one-based indexing

```

values = (1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0)
columns = (1 2 2 3)
rowIndex = (1 3 4 6)

```

zero-based indexing

```

values = (1 0 2 1 6 7 8 2 1 4 5 1 4 3 0 0 7 2 0 0)
columns = (0 1 1 2)
rowIndex = (0 2 3 5)

```

When storing symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For example, consider the symmetric sparse matrix *F*:

$$F = \begin{bmatrix} 1 & 0 & 6 & 7 & * & * \\ 2 & 1 & 8 & 2 & * & * \\ 6 & 8 & 1 & 4 & * & * \\ 7 & 2 & 5 & 2 & * & * \\ * & * & * & * & 7 & 2 \\ * & * & * & * & 0 & 0 \end{bmatrix}$$

If the size of the block equals 2, then the sparse matrix F can be represented as a 3×3 block matrix G with the following structure:

$$G = \begin{bmatrix} L & M & * \\ M' & N & * \\ * & * & Q \end{bmatrix}$$

where

$$L = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}, M = \begin{bmatrix} 6 & 7 \\ 8 & 2 \end{bmatrix}, M' = \begin{bmatrix} 6 & 8 \\ 7 & 2 \end{bmatrix}, N = \begin{bmatrix} 1 & 4 \\ 5 & 2 \end{bmatrix}, Q = \begin{bmatrix} 7 & 2 \\ 0 & 0 \end{bmatrix}$$

The symmetric matrix F can be represented in this 3-array variation of the BSR format (storing only the upper triangular part) as follows:

one-based indexing

```
values = (1 2 0 1 6 8 7 2 1 5 4 2 7 0 2 0)
columns = (1 2 2 3)
rowIndex = (1 3 4 5)
```

zero-based indexing

```
values = (1 0 2 1 6 7 8 2 1 4 5 2 7 2 0 0)
columns = (0 1 1 2)
rowIndex = (0 2 3 4)
```


Routine and Function Arguments

The major arguments in the BLAS routines are vector and matrix, whereas VML functions work on vector arguments only. The sections that follow discuss each of these arguments and provide examples.

Vector Arguments in BLAS

Vector arguments are passed in one-dimensional arrays. The array dimension (length) and vector increment are passed as integer variables. The length determines the number of elements in the vector. The increment (also called stride) determines the spacing between vector elements and the order of the elements in the array in which the vector is passed.

A vector of length n and increment $incx$ is passed in a one-dimensional array x whose values are defined as

```
x(1), x(1+|incx|), ..., x(1+(n-1)* |incx|)
```

If $incx$ is positive, then the elements in array x are stored in increasing order. If $incx$ is negative, the elements in array x are stored in decreasing order with the first element defined as $x(1+(n-1)* |incx|)$. If $incx$ is zero, then all elements of the vector have the same value, $x(1)$. The dimension of the one-dimensional array that stores the vector must always be at least

```
idimx = 1 + (n-1)* |incx|
```

Example. One-dimensional Real Array

Let $x(1:7)$ be the one-dimensional real array

```
x = (1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0).
```

If $incx = 2$ and $n = 3$, then the vector argument with elements in order from first to last is $(1.0, 5.0, 9.0)$.

If $incx = -2$ and $n = 4$, then the vector elements in order from first to last is $(13.0, 9.0, 5.0, 1.0)$.

If $incx = 0$ and $n = 4$, then the vector elements in order from first to last is $(1.0, 1.0, 1.0, 1.0)$.

One-dimensional substructures of a matrix, such as the rows, columns, and diagonals, can be passed as vector arguments with the starting address and increment specified. In Fortran, storing the m -by- n matrix is based on column-major ordering where the increment between elements in the same column is 1, the increment between elements in the same row is m , and the increment between elements on the same diagonal is $m + 1$.

Example. Two-dimensional Real Matrix

Let a be the real 5×4 matrix declared as `REAL A (5, 4)`.

To scale the third column of a by 2.0, use the BLAS routine `sscal` with the following calling sequence:

```
callsscal (5, 2.0, a(1,3), 1)
```

To scale the second row, use the statement:

```
callsscal (4, 2.0, a(2,1), 5)
```

To scale the main diagonal of A by 2.0, use the statement:

```
callsscal (5, 2.0, a(1,1), 6)
```

NOTE

The default vector argument is assumed to be 1.

Vector Arguments in VML

Vector arguments of VML mathematical functions are passed in one-dimensional arrays with unit vector increment. It means that a vector of length n is passed contiguously in an array a whose values are defined as $a[0]$, $a[1]$, ..., $a[n-1]$ (for the C interface).

To accommodate for arrays with other increments, or more complicated indexing, VML contains auxiliary pack/unpack functions that gather the array elements into a contiguous vector and then scatter them after the computation is complete.

Generally, if the vector elements are stored in a one-dimensional array a as

```
a[m0], a[m1], ..., a[mn-1]
```

and need to be regrouped into an array y as

```
y[k0], y[k1], ..., y[kn-1],
```

VML pack/unpack functions can use one of the following indexing methods:

Positive Increment Indexing

```
kj = incy * j, mj = inca * j, j = 0 ,..., n-1
```

Constraint: $\text{incy} > 0$ and $\text{inca} > 0$.

For example, setting $\text{incy} = 1$ specifies gathering array elements into a contiguous vector.

This method is similar to that used in BLAS, with the exception that negative and zero increments are not permitted.

Index Vector Indexing

```
kj = iy[j], mj = ia[j], j = 0 ,..., n-1,
```

where ia and iy are arrays of length n that contain index vectors for the input and output arrays a and y , respectively.

Mask Vector Indexing

Indices kj , mj are such that:

```
my[kj] ≠ 0, ma[mj] ≠ 0 , j = 0,..., n-1,
```

where ma and my are arrays that contain mask vectors for the input and output arrays a and y , respectively.

Matrix Arguments

Matrix arguments of the Intel® Math Kernel Library routines can be stored in either one- or two-dimensional arrays, using the following storage schemes:

- [conventional full storage](#) (in a two-dimensional array)
- [packed storage](#) for Hermitian, symmetric, or triangular matrices (in a one-dimensional array)
- [band storage](#) for band matrices (in a two-dimensional array)
- [rectangular full packed storage](#) for symmetric, Hermitian, or triangular matrices as compact as the Packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels.

Full storage is the simplest scheme. A matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} stored in the array element $a(i, j)$, where lda is the leading dimension of array a .

If a matrix is triangular (upper or lower, as specified by the argument $uplo$), only the elements of the relevant triangle are stored; the remaining elements of the array need not be set.

Routines that handle symmetric or Hermitian matrices allow for either the upper or lower triangle of the matrix to be stored in the corresponding elements of the array:

if $uplo = 'U'$, a_{ij} is stored as described for $i \leq j$, other elements of a need not be set.

if $uplo = 'L'$, a_{ij} is stored as described for $j \leq i$, other elements of a need not be set.

Packed storage allows you to store symmetric, Hermitian, or triangular matrices more compactly: the relevant triangle (again, as specified by the argument $uplo$) is packed by columns in a one-dimensional array ap :

if $uplo = 'U'$, a_{ij} is stored in $ap(i + j(j - 1)/2)$ for $i \leq j$

if $uplo = 'L'$, a_{ij} is stored in $ap(i + (2*n - j)*(j - 1)/2)$ for $j \leq i$.

In descriptions of LAPACK routines, arrays with packed matrices have names ending in p .

Band storage is as follows: an m -by- n band matrix with k_l non-zero sub-diagonals and k_u non-zero super-diagonals is stored compactly in a two-dimensional array ab with k_l+k_u+1 rows and n columns. Columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in rows of the array. Thus,

a_{ij} is stored in $ab(k_u+1+i-j, j)$ for $\max(1, j-k_u) \leq i \leq \min(n, j+k_l)$.

Use the band storage scheme only when k_l and k_u are much less than the matrix size n . Although the routines work correctly for all values of k_l and k_u , using the band storage is inefficient if your matrices are not really banded.

The band storage scheme is illustrated by the following example, when

$m = n = 6$, $k_l = 2$, $k_u = 1$

Array elements marked * are not used by the routines:

Banded matrix A						Band storage of A					
$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{bmatrix}$						*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}
						a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}
						a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	*
						a_{31}	a_{42}	a_{53}	a_{64}	*	*

When a general band matrix is supplied for *LU factorization*, space must be allowed to store k_l additional super-diagonals generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with $k_l + k_u$ super-diagonals. Thus,

a_{ij} is stored in $ab(k_l+k_u+1+i-j, j)$ for $\max(1, j-k_u) \leq i \leq \min(n, j+k_l)$.

The band storage scheme for LU factorization is illustrated by the following example, when $m = n = 6$, $k_l = 2$, $k_u = 1$:

Banded matrix A						Band storage of A					
a_{11}	a_{12}	0	0	0	0	*	*	*	+	+	+
a_{21}	a_{22}	a_{23}	0	0	0	*	*	+	+	+	+
a_{31}	a_{32}	a_{33}	a_{34}	0	0	*	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}
0	a_{42}	a_{43}	a_{44}	a_{45}	0	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}
0	0	a_{53}	a_{54}	a_{55}	a_{56}	a_{21}	a_{32}	a_{43}	a_{54}	a_{65}	*
0	0	0	a_{64}	a_{65}	a_{66}	a_{31}	a_{42}	a_{53}	a_{64}	*	*

Array elements marked * are not used by the routines; elements marked + need not be set on entry, but are required by the LU factorization routines to store the results. The input array will be overwritten on exit by the details of the LU factorization as follows:

100 | 100
100 | 100
100 | 100

where u_{ij} are the elements of the upper triangular matrix U , and m_{ij} are the multipliers used during factorization.

Triangular band matrices are stored in the same format, with either $kl = 0$ if upper triangular, or $ku = 0$ if lower triangular. For symmetric or Hermitian band matrices with k sub-diagonals or super-diagonals, you need to store only the upper or lower triangle, as specified by the argument *uplo*:

if $uplo = 'U'$, a_{ij} is stored in $ab(k+1+i-j, j)$ for $\max(1, j-k) \leq i \leq j$

if $uplo = 'L'$, $a_{i,j}$ is stored in $ab(1+i-j,j)$ for $j \leq i \leq \min(n, j+k)$.

In descriptions of LAPACK routines, arrays that hold matrices in band storage have names ending in *b*.

In Fortran, column-major ordering of storage is assumed. This means that elements of the same column occupy successive storage locations.

Three quantities are usually associated with a two-dimensional array argument: its leading dimension, which specifies the number of storage locations between elements in the same row, its number of rows, and its number of columns. For a matrix in full storage, the leading dimension of the array must be at least as large as the number of rows in the matrix.

A character transposition parameter is often passed to indicate whether the matrix argument is to be used in normal or transposed form or, for a complex matrix, if the conjugate transpose of the matrix is to be used.

The values of the transposition parameter for these three cases are the following:

'N' or 'n' normal (no conjugation, no transposition)

'T' or 't' transpose

'C' or 'c' conjugate transpose.

Example. Two-Dimensional Complex Array

Suppose A (1:5, 1:4) is the complex two-dimensional array presented by matrix

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) & (1.3, 0.13) & (1.4, 0.14) \\ (2.1, 0.21) & (2.2, 0.22) & (2.3, 0.23) & (1.4, 0.24) \\ (3.1, 0.31) & (3.2, 0.32) & (3.3, 0.33) & (1.4, 0.34) \\ (4.1, 0.41) & (4.2, 0.42) & (4.3, 0.43) & (1.4, 0.44) \\ (5.1, 0.51) & (5.2, 0.52) & (5.3, 0.53) & (1.4, 0.54) \end{bmatrix}$$

Let $transa$ be the transposition parameter, m be the number of rows, n be the number of columns, and lda be the leading dimension. Then if

$transa = 'N'$, $m = 4$, $n = 2$, and $lda = 5$, the matrix argument would be

$$\begin{bmatrix} (1.1, 0.11) & (1.2, 0.12) \\ (2.1, 0.21) & (2.2, 0.22) \\ (3.1, 0.31) & (3.2, 0.32) \\ (4.1, 0.41) & (4.2, 0.42) \end{bmatrix}$$

If $transa = 'T'$, $m = 4$, $n = 2$, and $lda = 5$, the matrix argument would be

$$\begin{bmatrix} \dots & \boxed{(1.1, 0.11)} & \dots & \dots & \dots \\ \dots & \boxed{(1.2, 0.12)} & \dots & \dots & \dots \end{bmatrix}$$

If $transa = 'C'$, $m = 4$, $n = 2$, and $lda = 5$, the matrix argument would be

$$\begin{bmatrix} (1.1, -0.11) & (2.1, -0.21) & (3.1, -0.31) & (4.1, -0.41) \\ (1.2, -0.12) & (2.2, -0.22) & (3.2, -0.32) & (4.2, -0.42) \end{bmatrix}$$

Note that care should be taken when using a leading dimension value which is different from the number of rows specified in the declaration of the two-dimensional array. For example, suppose the array A above is declared as `COMPLEX A (5,4)`.

Then if $transa = 'N'$, $m = 3$, $n = 4$, and $lda = 4$, the matrix argument will be

$$\begin{bmatrix} (1.1, 0.11) & (5.1, 0.51) & (4.2, 0.42) & (3.3, 0.33) \\ (2.1, 0.21) & (1.2, 0.12) & (5.2, 0.52) & (4.3, 0.43) \\ (3.1, 0.31) & (2.2, 0.22) & (1.3, 0.13) & (5.3, 0.53) \end{bmatrix}$$

Rectangular Full Packed storage allows you to store symmetric, Hermitian, or triangular matrices as compact as the Packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels. To store an n -by- n triangle (and suppose for simplicity that n is even), you partition the triangle into three parts: two $n/2$ -by- $n/2$ triangles and an $n/2$ -by- $n/2$ square, then pack this as an n -by- $n/2$ rectangle (or $n/2$ -by- n rectangle), by transposing (or transpose-conjugating) one of the triangles and packing it next to the other triangle. Since the two triangles are stored in full storage, you can use existing efficient routines on them.

There are eight cases of RFP storage representation: when n is even or odd, the packed matrix is transposed or not, the triangular matrix is lower or upper. See below for all the eight storage schemes illustrated:

n is odd, A is lower triangular

Full format	RFP (not transposed)	RFP (transposed)
$a_{11} \quad x \quad x \quad x \quad x \quad x \quad x$	$a_{11} \quad a_{55} \quad a_{65} \quad a_{75}$	
$a_{21} \quad a_{22} \quad x \quad x \quad x \quad x \quad x$	$a_{21} \quad a_{22} \quad a_{66} \quad a_{76}$	
$a_{31} \quad a_{32} \quad a_{33} \quad x \quad x \quad x \quad x$	$a_{31} \quad a_{32} \quad a_{33} \quad a_{77}$	
$a_{41} \quad a_{42} \quad a_{43} \quad a_{44} \quad x \quad x \quad x$	$a_{41} \quad a_{42} \quad a_{43} \quad a_{44}$	
$\mathbf{a}_{51} \quad \mathbf{a}_{52} \quad \mathbf{a}_{53} \quad \mathbf{a}_{54} \quad a_{55} \quad x \quad x$	$\mathbf{a}_{51} \quad \mathbf{a}_{52} \quad \mathbf{a}_{53} \quad \mathbf{a}_{54}$	
$\mathbf{a}_{61} \quad \mathbf{a}_{62} \quad \mathbf{a}_{63} \quad \mathbf{a}_{64} \quad a_{65} \quad a_{66} \quad x$	$\mathbf{a}_{61} \quad \mathbf{a}_{62} \quad \mathbf{a}_{63} \quad \mathbf{a}_{64}$	
$\mathbf{a}_{71} \quad \mathbf{a}_{72} \quad \mathbf{a}_{73} \quad \mathbf{a}_{74} \quad a_{75} \quad a_{76} \quad a_{77}$	$\mathbf{a}_{71} \quad \mathbf{a}_{72} \quad \mathbf{a}_{73} \quad \mathbf{a}_{74}$	

n is even, A is lower triangular

Full format	RFP (not transposed)	RFP (transposed)
$a_{11} \quad x \quad x \quad x \quad x \quad x$	$a_{44} \quad a_{54} \quad a_{64}$	
$a_{21} \quad a_{22} \quad x \quad x \quad x \quad x$	$a_{11} \quad a_{55} \quad a_{65}$	
$a_{31} \quad a_{32} \quad a_{33} \quad x \quad x \quad x$	$a_{21} \quad a_{22} \quad a_{66}$	
$\mathbf{a}_{41} \quad \mathbf{a}_{42} \quad \mathbf{a}_{43} \quad a_{44} \quad x \quad x$	$\mathbf{a}_{31} \quad a_{41} \quad \mathbf{a}_{51} \quad \mathbf{a}_{61}$	
$\mathbf{a}_{51} \quad \mathbf{a}_{52} \quad \mathbf{a}_{53} \quad a_{54} \quad a_{55} \quad x$	$\mathbf{a}_{54} \quad a_{55} \quad a_{22} \quad \mathbf{a}_{42} \quad \mathbf{a}_{52} \quad \mathbf{a}_{62}$	
$\mathbf{a}_{61} \quad \mathbf{a}_{62} \quad \mathbf{a}_{63} \quad a_{64} \quad a_{65} \quad a_{66}$	$\mathbf{a}_{41} \quad \mathbf{a}_{42} \quad \mathbf{a}_{43} \quad a_{64} \quad a_{65} \quad a_{66}$	
	$\mathbf{a}_{51} \quad \mathbf{a}_{52} \quad \mathbf{a}_{53}$	
	$\mathbf{a}_{61} \quad \mathbf{a}_{62} \quad \mathbf{a}_{63}$	

n is odd, A is upper triangular

Full format	RFP (not transposed)	RFP (transposed)
$a_{11} \quad a_{12} \quad a_{13} \quad \mathbf{a}_{14} \quad \mathbf{a}_{15} \quad \mathbf{a}_{16} \quad a_{17}$	$\mathbf{a}_{14} \quad \mathbf{a}_{15} \quad \mathbf{a}_{16} \quad \mathbf{a}_{17}$	
$x \quad a_{22} \quad a_{23} \quad \mathbf{a}_{24} \quad \mathbf{a}_{25} \quad \mathbf{a}_{26} \quad a_{27}$	$\mathbf{a}_{24} \quad \mathbf{a}_{25} \quad \mathbf{a}_{26} \quad \mathbf{a}_{27}$	
$x \quad x \quad a_{33} \quad \mathbf{a}_{34} \quad \mathbf{a}_{35} \quad \mathbf{a}_{36} \quad a_{37}$	$\mathbf{a}_{34} \quad \mathbf{a}_{35} \quad \mathbf{a}_{36} \quad \mathbf{a}_{37}$	
$x \quad x \quad x \quad a_{44} \quad a_{45} \quad a_{46} \quad a_{47}$	$a_{44} \quad a_{45} \quad a_{46} \quad a_{47}$	
$x \quad x \quad x \quad x \quad a_{55} \quad a_{56} \quad a_{57}$	$a_{11} \quad a_{55} \quad a_{56} \quad a_{57}$	
$x \quad x \quad x \quad x \quad x \quad a_{66} \quad a_{67}$	$a_{12} \quad a_{22} \quad a_{66} \quad a_{67}$	
$x \quad x \quad x \quad x \quad x \quad x \quad a_{77}$	$a_{13} \quad a_{23} \quad a_{33} \quad a_{77}$	

n is even, A is upper triangular

Full format	RFP (not transposed)	RFP (transposed)
$a_{11} \ a_{12} \ a_{13} \ a_{14} \ a_{15} \ a_{16}$	$\mathbf{a}_{14} \ \mathbf{a}_{15} \ \mathbf{a}_{16}$	
$\mathbb{X} \ a_{22} \ a_{23} \ a_{24} \ a_{25} \ a_{26}$	$\mathbf{a}_{24} \ \mathbf{a}_{25} \ \mathbf{a}_{26}$	
$\mathbb{X} \ \mathbb{X} \ a_{33} \ a_{34} \ a_{35} \ a_{36}$	$\mathbf{a}_{34} \ \mathbf{a}_{35} \ \mathbf{a}_{36}$	$\mathbf{a}_{14} \ \mathbf{a}_{24} \ \mathbf{a}_{34} \ a_{44} \ a_{11} \ a_{12} \ a_{13}$
$\mathbb{X} \ \mathbb{X} \ a_{44} \ a_{45} \ a_{46}$	$a_{44} \ a_{45} \ a_{46}$	$\mathbf{a}_{15} \ \mathbf{a}_{25} \ \mathbf{a}_{35} \ a_{45} \ a_{55} \ a_{22} \ a_{23}$
$\mathbb{X} \ \mathbb{X} \ \mathbb{X} \ a_{55} \ a_{56}$	$a_{11} \ a_{55} \ a_{56}$	$\mathbf{a}_{16} \ \mathbf{a}_{26} \ \mathbf{a}_{36} \ a_{46} \ a_{56} \ a_{66} \ a_{33}$
$\mathbb{X} \ \mathbb{X} \ \mathbb{X} \ \mathbb{X} \ a_{66}$	$a_{12} \ a_{22} \ a_{66}$	
	$a_{13} \ a_{23} \ a_{33}$	

Intel MKL provides a number of routines such as [?hfrk](#), [?sfrk](#) performing BLAS operations working directly on RFP matrices, as well as some conversion routines, for instance, [?pttf](#) goes from the standard packed format to RFP and [?trttf](#) goes from the full format to RFP.

Please refer to the Netlib site for more information.

Note that in the descriptions of LAPACK routines, arrays with RFP matrices have names ending in `fp`.

Specific Features of Fortran 95 Interfaces for LAPACK Routines

C

Intel® MKL implements Fortran 95 interface for LAPACK package, further referred to as MKL LAPACK95, to provide full capacity of MKL FORTRAN 77 LAPACK routines. This is the principal difference of Intel MKL from the Netlib Fortran 95 implementation for LAPACK.

A new feature of MKL LAPACK95 by comparison with Intel MKL LAPACK77 implementation is presenting a package of source interfaces along with wrappers that make the implementation compiler-independent. As a result, the MKL LAPACK package can be used in all programming environments intended for Fortran 95.

Depending on the degree and type of difference from Netlib implementation, the MKL LAPACK95 interfaces fall into several groups that require different transformations (see "[MKL Fortran 95 Interfaces for LAPACK Routines vs. Netlib Implementation](#)"). The groups are given in full with the calling sequences of the routines and appropriate differences from Netlib analogs.

The following conventions are used:

```
<interface> ::= <name of interface> '(' <arguments list> ')'
<arguments list> ::= <first argument> {<argument>}*
<first argument> ::= <identifier>
<argument> ::= <required argument>|<optional argument>
<required argument> ::= ',' <identifier>
<optional argument> ::= '[' ,<identifier> ']'
<name of interface> ::= <identifier>
```

where defined notions are separated from definitions by ::=, notion names are marked by angle brackets, terminals are given in quotes, and {...}* denotes repetition zero, one, or more times.

<first argument> and each <required argument> should be present in all calls of denoted interface, <optional argument> may be omitted. Comments to interface definitions are provided where necessary. Comment lines begin with character !.

Two interfaces with one name are presented when two variants of subroutine calls (separated by types of arguments) exist.

Interfaces Identical to Netlib

```
GERFS(A,AF,IPIV,B,X[,TRANS][,FERR][,BERR][,INFO])
GETRI(A,IPIV[,INFO])
GEEQU(A,R,C[,ROWCND][,COLCND][,AMAX][,INFO])
GESV(A,B[,IPIV][,INFO])
GESVX(A,B,X[,AF][,IPIV][,FACT][,TRANS][,EQUED][,R][,C][,FERR][,BERR]
[,RCOND][,RPVGRW][,INFO])
GTSV(DL,D,DU,B[,INFO])
GTSVX(DL,D,DU,B,X[,DLF][,DF][,DUF][,DU2][,IPIV][,FACT][,TRANS][,FERR]
[,BERR][,RCOND][,INFO])
POSV(A,B[,UPLO][,INFO])
POSVX(A,B,X[,UPLO][,AF][,FACT][,EQUED][,S][,FERR][,BERR][,RCOND][,INFO])
PTSV(D,E,B[,INFO])
PTSVX(D,E,B,X[,DF][,EF][,FACT][,FERR][,BERR][,RCOND][,INFO])
SYSV(A,B[,UPLO][,IPIV][,INFO])
SYSVX(A,B,X[,UPLO][,AF][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO])
HESVX(A,B,X[,UPLO][,AF][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO])
HESV(A,B[,UPLO][,IPIV][,INFO])
SPSV(AP,B[,UPLO][,IPIV][,INFO])
HPSV(AP,B[,UPLO][,IPIV][,INFO])
SYTRD(A,TAU[,UPLO][,INFO])
```

```

ORGTR (A, TAU[, UPLO][, INFO])
HETRD (A, TAU[, UPLO][, INFO])
UNGTR (A, TAU[, UPLO][, INFO])
SYGST (A, B[, ITYPE][, UPLO][, INFO])
HEGST (A, B[, ITYPE][, UPLO][, INFO])
GELS (A, B[, TRANS][, INFO])
GELSY (A, B[, RANK][, JPVT][, RCOND][, INFO])
GELSS (A, B[, RANK][, S][, RCOND][, INFO])
GELSD (A, B[, RANK][, S][, RCOND][, INFO])
GGLSE (A, B, C, D, X[, INFO])
GGGLM (A, B, D, X, Y[, INFO])
SYEV (A, W[, JOBZ][, UPLO][, INFO])
HEEV (A, W[, JOBZ][, UPLO][, INFO])
SYEVD (A, W[, JOBZ][, UPLO][, INFO])
HEEVD (A, W[, JOBZ][, UPLO][, INFO])
STEV (D, E[, Z][, INFO])
STEVD (D, E[, Z][, INFO])
STEVX (D, E, W[, Z][, VL][, VU][, IL][, IU][, M][, IFAIL][, ABSTOL][, INFO])
STEVR (D, E, W[, Z][, VL][, VU][, IL][, IU][, M][, ISUPPZ][, ABSTOL][, INFO])
GEES (A, WR, WI[, VS][, SELECT][, SDIM][, INFO])
GEES (A, W[, VS][, SELECT][, SDIM][, INFO])
GEESX (A, WR, WI[, VS][, SELECT][, SDIM][, RCONDE][, RCONDV][, INFO])
GEESX (A, W[, VS][, SELECT][, SDIM][, RCONDE][, RCONDV][, INFO])
GEEV (A, WR, WI[, VL][, VR][, INFO])
GEEV (A, W[, VL][, VR][, INFO])
GEEVX (A, WR, WI[, VL][, VR][, BALANC][, ILO][, IHII][, SCALE][, ABNRM][, RCONDE][, RCONDV][, INFO])
GEEVX (A, W[, VL][, VR][, BALANC][, ILO][, IHII][, SCALE][, ABNRM][, RCONDE]
[, RCONDV][, INFO])
GESVD (A, S[, U][, VT][, WW][, JOB][, INFO])
GGSVD (A, B, ALPHA, BETA[, K][, L][, U][, V][, Q][, IWORK][, INFO])
SYGV (A, B, W[, ITYPE][, JOBZ][, UPLO][, INFO])
HEGV (A, B, W[, ITYPE][, JOBZ][, UPLO][, INFO])
SYGVD (A, B, W[, ITYPE][, JOBZ][, UPLO][, INFO])
HEGVD (A, B, W[, ITYPE][, JOBZ][, UPLO][, INFO])
SPGVD (AP, BP, W[, ITYPE][, UPLO][, Z][, INFO])
HPGVD (AP, BP, W[, ITYPE][, UPLO][, Z][, INFO])
SPGVX (AP, BP, W[, ITYPE][, UPLO][, Z][, VL][, VU][, IL][, IU][, M][, IFAIL][, ABSTOL][, INFO])
HPGVX (AP, BP, W[, ITYPE][, UPLO][, Z][, VL][, VU][, IL][, IU][, M][, IFAIL][, ABSTOL][, INFO])
SBGVD (AB, BB, W[, UPLO][, Z][, INFO])
HBGVD (AB, BB, W[, UPLO][, Z][, INFO])
SBGVX (AB, BB, W[, UPLO][, Z][, VL][, VU][, IL][, IU][, M][, IFAIL][, Q][, ABSTOL][, INFO])
HBGVX (AB, BB, W[, UPLO][, Z][, VL][, VU][, IL][, IU][, M][, IFAIL][, Q][, ABSTOL][, INFO])
GGES (A, B, ALPHAR, ALPHAI, BETA[, VSL][, VSR][, SELECT][, SDIM][, INFO])
GGES (A, B, ALPHA, BETA[, VSL][, VSR][, SELECT][, SDIM][, INFO])
GGESX (A, B, ALPHAR, ALPHAI, BETA[, VSL][, VSR][, SELECT][, SDIM][, RCONDE][, RCONDV][, INFO])
GGEV (A, B, ALPHAR, ALPHAI, BETA[, VL][, VR][, INFO])
GGEV (A, B, ALPHA, BETA[, VL][, VR][, INFO])
GGEVX (A, B, ALPHAR, ALPHAI, BETA[, VL][, VR][, BALANC][, ILO][, IHII][, LSCALE][, RSCALE][, ABNRM]
[, BBNRM][, RCONDE][, RCONDV][, INFO])
GGEVX (A, B, ALPHA, BETA[, VL][, VR][, BALANC][, ILO][, IHII][, LSCALE][, RSCALE][, ABNRM]
[, BBNRM][, RCONDE][, RCONDV][, INFO])

```

Interfaces with Replaced Argument Names

Argument names in the routines of this group are replaced as follows:

Netlib Argument Name	MKL Argument Name
A	AB
A	AP

Netlib Argument Name	MKL Argument Name
AF	AFB
AF	AFP
B	BB
B	BP
K	KL
GBSV(AB,B[,KL][,IPIV][,INFO]) ! netlib: (A,B,K,IPIV,INFO)	
GBSVX(AB,B,X[,KL][,AFB][,IPIV][,FACT][,TRANS][,EQUED][,R][,C][,FERR] [,BERR][,RCOND][,RPVGRW][,INFO]) ! netlib: (A,B,X,KL,AF,IPIV,FACT,TRANS,EQUED,R,C,FERR, ! BERR,RCOND,RPVGRW,INFO)	
PPSV(AP,B[,UPLO][,INFO]) ! netlib: (A,B,UPLO,INFO)	
PPSVX(AP,B,X[,UPLO][,AFB][,FACT][,EQUED][,S][,FERR][,BERR][,RCOND][,INFO]) ! netlib: (A,B,X,UPLO,AF,FACT,EQUED,S,FERR,BERR,RCOND,INFO) !	
PBSV(AB,B[,UPLO][,INFO]) ! netlib: (A,B,UPLO,INFO)	
PBSVX(AB,B,X[,UPLO][,AFB][,FACT][,EQUED][,S][,FERR][,BERR][,RCOND][,INFO]) ! netlib: (A,B,X,UPLO,AF,FACT,EQUED,S,FERR,BERR,RCOND,INFO) !	
SPSV(AB,B,X[,UPLO][,AFB][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO]) ! netlib: (A,B,X,UPLO,AF,IPIV,FACT,FERR,BERR,RCOND,INFO)	
HPSVX(AP,B,X[,UPLO][,AFB][,IPIV][,FACT][,FERR][,BERR][,RCOND][,INFO]) ! netlib: (A,B,X,UPLO,AF,IPIV,FACT,FERR,BERR,RCOND,INFO)	
SPEV(AP,W[,UPLO][,Z][,INFO]) ! netlib: (A,W,UPLO,Z,INFO)	
HPEV(AP,W[,UPLO][,Z][,INFO]) ! netlib: (A,W,UPLO,Z,INFO)	
SPEVD(AP,W[,UPLO][,Z][,INFO]) ! netlib: (A,W,UPLO,Z,INFO)	
HPEVD(AP,W[,UPLO][,Z][,INFO]) ! netlib: (A,W,UPLO,Z,INFO)	
SPEVX(AP,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO]) ! netlib: (A,B,W,UPLO,Z,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)	
HPEVX(AP,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO]) ! netlib: (A,B,W,UPLO,Z,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)	
SBEV(AB,W[,UPLO][,Z][,INFO]) ! netlib: (A,W,UPLO,Z,INFO)	
HBEV(AB,W[,UPLO][,Z][,INFO]) ! netlib: (A,W,UPLO,Z,INFO)	
SBEVD(AB,W[,UPLO][,Z][,INFO]) ! netlib: (A,W,UPLO,Z,INFO)	

```
HBEVD(AB,W[,UPLO][,Z][,INFO])
!    netlib: (A,W,UPLO,Z,INFO)
```

```
SBEVX(AB,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,Q][,ABSTOL][,INFO])
!    netlib: (A,B,W,UPLO,Z,VL,VU,IL,IU,M,IFAIL,Q,ABSTOL,INFO)
```

```
HBEVX(AB,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,Q][,ABSTOL][,INFO])
!    netlib: (A,B,W,UPLO,Z,VL,VU,IL,IU,M,IFAIL,Q,ABSTOL,INFO)
```

```
SPGV(AP,BP,W[,ITYPE][,UPLO][,Z][,INFO])
!    netlib: (A,B,W,ITYPE,UPLO,Z,INFO)
```

```
HPGV(AB,BP,W[,ITYPE][,UPLO][,Z][,INFO])
!    netlib: (A,B,W,ITYPE,UPLO,Z,INFO)
```

```
SBGV(AB,BB,W[,UPLO][,Z][,INFO])
!    netlib: (A,B,W,UPLO,Z,INFO)
```

```
HBGV(AB,BB,W[,UPLO][,Z][,INFO])
!    netlib: (A,B,W,UPLO,Z,INFO)
```

Modified Netlib Interfaces

```
SYEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!    Interface netlib95 exists, parameters:
!    netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!    Different order for parameter UPLO, netlib: 4, mkl: 3
!    Absent mkl parameter: JOBZ
!    Extra mkl parameter: Z
```

```
HEEVX(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!    Interface netlib95 exists, parameters:
!    netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!    Different order for parameter UPLO, netlib: 4, mkl: 3
!    Absent mkl parameter: JOBZ
!    Extra mkl parameter: Z
```

```
SYEVR(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,ISUPPZ][,ABSTOL][,INFO])
!    Interface netlib95 exists, parameters:
!    netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,ISUPPZ,ABSTOL,INFO)
!    Different order for parameter UPLO, netlib: 4, mkl: 3
!    Absent mkl parameter: JOBZ
!    Extra mkl parameter: Z
```

```
HEEVR(A,W[,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,ISUPPZ][,ABSTOL][,INFO])
!    Interface netlib95 exists, parameters:
!    netlib: (A,W,JOBZ,UPLO,VL,VU,IL,IU,M,ISUPPZ,ABSTOL,INFO)
!    Different order for parameter UPLO, netlib: 4, mkl: 3
!    Absent mkl parameter: JOBZ
!    Extra mkl parameter: Z
```

```
GESDD(A,S[,U][,VT][,JOBZ][,INFO])
!    Interface netlib95 exists, parameters:
!    netlib: (A,S,U,VT,WW,JOB,INFO)
!    Different number for parameter, netlib: 7, mkl: 6
!    Absent mkl parameter: WW
```

```
!  Absent mkl parameter: JOB
!  Different order for parameter INFO, netlib: 7, mkl: 6
!  Extra mkl parameter: JOBZ
```

```
SYGVX(A,B,W[,ITYPE][,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!  Interface netlib95 exists, parameters:
!  netlib: (A,B,W,ITYPE,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!  Different order for parameter UPLO, netlib: 6, mkl: 5
!  Absent mkl parameter: JOBZ
!  Extra mkl parameter: Z
```

```
HEGVX(A,B,W[,ITYPE][,UPLO][,Z][,VL][,VU][,IL][,IU][,M][,IFAIL][,ABSTOL][,INFO])
!  Interface netlib95 exists, parameters:
!  netlib: (A,B,W,ITYPE,JOBZ,UPLO,VL,VU,IL,IU,M,IFAIL,ABSTOL,INFO)
!  Different order for parameter UPLO, netlib: 6, mkl: 5
!  Absent mkl parameter: JOBZ
!  Extra mkl parameter: Z
```

```
GETRS(A,IPIV,B[,TRANS][,INFO])
!  Interface netlib95 exists:
!  Different intents for parameter A, netlib: INOUT, mkl: IN
```

Interfaces Absent From Netlib

```
GTTRF(DL,D,DU,DU2[,IPIV][,INFO])
PPTRF(A[,UPLO][,INFO])
PBTRF(A[,UPLO][,INFO])
PTTRF(D,E[,INFO])
SYTRF(A[,UPLO][,IPIV][,INFO])
HETRF(A[,UPLO][,IPIV][,INFO])
SPTRF(A[,UPLO][,IPIV][,INFO])
HPTRF(A[,UPLO][,IPIV][,INFO])
GBTRS(A,B,IPIV[,KL][,TRANS][,INFO])
GTTRS(DL,D,DU,DU2,B,IPIV[,TRANS][,INFO])
POTRS(A,B[,UPLO][,INFO])
PPTRS(A,B[,UPLO][,INFO])
PBTRS(A,B[,UPLO][,INFO])
PTTRS(D,E,B[,INFO])
PTTRS(D,E,B[,UPLO][,INFO])
SYTRS(A,B,IPIV[,UPLO][,INFO])
HETRS(A,B,IPIV[,UPLO][,INFO])
SPTRS(A,B,IPIV[,UPLO][,INFO])
HPTRS(A,B,IPIV[,UPLO][,INFO])
TRTRS(A,B[,UPLO][,TRANS][,DIAG][,INFO])
TPTRS(A,B[,UPLO][,TRANS][,DIAG][,INFO])
TBTRS(A,B[,UPLO][,TRANS][,DIAG][,INFO])
GECON(A,ANORM,RCOND[,NORM][,INFO])
GBCON(A,IPIV,ANORM,RCOND[,KL][,NORM][,INFO])
GTCON(DL,D,DU,DU2,IPIV,ANORM,RCOND[,NORM][,INFO])
POCON(A,ANORM,RCOND[,UPLO][,INFO])
PPCON(A,ANORM,RCOND[,UPLO][,INFO])
PBCON(A,ANORM,RCOND[,UPLO][,INFO])
PTCON(D,E,ANORM,RCOND[,INFO])
SYCON(A,IPIV,ANORM,RCOND[,UPLO][,INFO])
HECON(A,IPIV,ANORM,RCOND[,UPLO][,INFO])
SPCON(A,IPIV,ANORM,RCOND[,UPLO][,INFO])
HPCON(A,IPIV,ANORM,RCOND[,UPLO][,INFO])
TRCON(A,RCOND[,UPLO][,DIAG][,NORM][,INFO])
TPCON(A,RCOND[,UPLO][,DIAG][,NORM][,INFO])
```

```
TBCON(A, RCOND[, UPLO][, DIAG][, NORM][, INFO])
GBRFS(A, AF, IPIV, B, X[, KL][, TRANS][, FERR][, BERR][, INFO])
GTRFS(DL, D, DU, DLF, DF, DUF, DU2, IPIV, B, X[, TRANS][, FERR][, BERR][, INFO])
PORFS(A, AF, B, X[, UPLO][, FERR][, BERR][, INFO])
PPRFS(A, AF, B, X[, UPLO][, FERR][, BERR][, INFO])
PBRFS(A, AF, B, X[, UPLO][, FERR][, BERR][, INFO])
PTRFS(D, DF, E, EF, B, X[, FERR][, BERR][, INFO])
PTRFS(D, DF, E, EF, B, X[, UPLO][, FERR][, BERR][, INFO])
SYRFS(A, AF, IPIV, B, X[, UPLO][, FERR][, BERR][, INFO])
HERFS(A, AF, IPIV, B, X[, UPLO][, FERR][, BERR][, INFO])
SPRFS(A, AF, IPIV, B, X[, UPLO][, FERR][, BERR][, INFO])
HPRFS(A, AF, IPIV, B, X[, UPLO][, FERR][, BERR][, INFO])
TRRFS(A, B, X[, UPLO][, TRANS][, DIAG][, FERR][, BERR][, INFO])
TPRFS(A, B, X[, UPLO][, TRANS][, DIAG][, FERR][, BERR][, INFO])
TBRFS(A, B, X[, UPLO][, TRANS][, DIAG][, FERR][, BERR][, INFO])
POTRI(A[, UPLO][, INFO])
PPTRI(A[, UPLO][, INFO])
SYTRI(A, IPIV[, UPLO][, INFO])
HETRI(A, IPIV[, UPLO][, INFO])
SPTRI(A, IPIV[, UPLO][, INFO])
HPTRI(A, IPIV[, UPLO][, INFO])
TRTRI(A[, UPLO][, DIAG][, INFO])
TPTRI(A[, UPLO][, DIAG][, INFO])
GBEQU(A, R, C[, KL][, ROWCND][, COLCND][, AMAX][, INFO])
POEQU(A, S[, SCOND][, AMAX][, INFO])
PPEQU(A, S[, SCOND][, AMAX][, UPLO][, INFO])
PBEQU(A, S[, SCOND][, AMAX][, UPLO][, INFO])
HESV(A, B[, UPLO][, IPIV][, INFO])
HPSV(A, B[, UPLO][, IPIV][, INFO])
GEQRF(A[, TAU][, INFO])
GEQPF(A, JPVT[, TAU][, INFO])
GEQP3(A, JPVT[, TAU][, INFO])
ORGQR(A, TAU[, INFO])
ORMQR(A, TAU, C[, SIDE][, TRANS][, INFO])
UNGQR(A, TAU[, INFO])
UNMQR(A, TAU, C[, SIDE][, TRANS][, INFO])
GEQLF(A[, TAU][, INFO])
ORGLQ(A, TAU[, INFO])
ORMLQ(A, TAU, C[, SIDE][, TRANS][, INFO])
UNGQLQ(A, TAU[, INFO])
UNMLQ(A, TAU, C[, SIDE][, TRANS][, INFO])
GEQLF(A[, TAU][, INFO])
ORGQL(A, TAU[, INFO])
UNGQL(A, TAU[, INFO])
ORMQL(A, TAU, C[, SIDE][, TRANS][, INFO])
UNMQL(A, TAU, C[, SIDE][, TRANS][, INFO])
GERQF(A[, TAU][, INFO])
ORGRO(A, TAU[, INFO])
UNGRO(A, TAU[, INFO])
ORMRQ(A, TAU, C[, SIDE][, TRANS][, INFO])
UNMRQ(A, TAU, C[, SIDE][, TRANS][, INFO])
TZRZF(A[, TAU][, INFO])
ORMRZ(A, TAU, C, L[, SIDE][, TRANS][, INFO])
UNMRZ(A, TAU, C, L[, SIDE][, TRANS][, INFO])
GGQRF(A, B[, TAUA][, TAUB][, INFO])
GGRQF(A, B[, TAUA][, TAUB][, INFO])
GEBRD(A[, D][, E][, TAUQ][, TAUP][, INFO])
GBBRD(A[, C][, D][, E][, Q][, PT][, KL][, M][, INFO])
ORGBR(A, TAU[, VECT][, INFO])
ORMBR(A, TAU, C[, VECT][, SIDE][, TRANS][, INFO])
```

```

ORMTR (A, TAU, C[, SIDE][, UPLO][, TRANS][, INFO])
UNGMR (A, TAU[, VECT][, INFO])
UNMBR (A, TAU, C[, VECT][, SIDE][, TRANS][, INFO])
BDSQR (D, E[, VT][, U][, C][, UPLO][, INFO])
BDSDC (D, E[, U][, VT][, Q][, IQ][, UPLO][, INFO])
UNMTR (A, TAU, C[, SIDE][, UPLO][, TRANS][, INFO])
SPTRD (A, TAU[, UPLO][, INFO])
OPGTR (A, TAU, Q[, UPLO][, INFO])
OPMTR (A, TAU, C[, SIDE][, UPLO][, TRANS][, INFO])
HPTRD (A, TAU[, UPLO][, INFO])
UPGTR (A, TAU, Q[, UPLO][, INFO])
UPMTR (A, TAU, C[, SIDE][, UPLO][, TRANS][, INFO])
SBTRD (A[, Q][, VECT][, UPLO][, INFO])
HBTRD (A[, Q][, VECT][, UPLO][, INFO])
STERF (D, E[, INFO])
STEQR (D, E[, Z][, COMPZ][, INFO])
STEDC (D, E[, Z][, COMPZ][, INFO])
STEGR (D, E, W[, Z][, VL][, VU][, IL][, IU][, M][, ISUPPZ][, ABSTOL][, INFO])
PTEQR (D, E[, Z][, COMPZ][, INFO])
STEBZ (D, E, M, NSPLIT, W, IBLOCK, ISPLIT[, ORDER][, VL][, VU][, IL][, IU][, ABSTOL][, INFO])
STEIN (D, E, W, IBLOCK, ISPLIT, Z[, IFAILV][, INFO])
DISNA (D, SEP[, JOB][, MINMN][, INFO])
SPGST (A, B[, ITYPE][, UPLO][, INFO])
HPGST (A, B[, ITYPE][, UPLO][, INFO])
SBGST (A, B[, X][, UPLO][, INFO])
HBGST (A, B[, X][, UPLO][, INFO])
PBSTF (B[, UPLO][, INFO])
GEHRD (A[, TAU][, ILO][, IHII][, INFO])
ORGHR (A, TAU[, ILO][, IHII][, INFO])
ORMHR (A, TAU, C[, ILO][, IHII][, SIDE][, TRANS][, INFO])
UNGHR (A, TAU[, ILO][, IHII][, INFO])
UNMHR (A, TAU, C[, ILO][, IHII][, SIDE][, TRANS][, INFO])
GEBAL (A[, SCALE][, ILO][, IHII][, JOB][, INFO])
GEBAK (V, SCALE[, ILO][, IHII][, JOB][, SIDE][, INFO])
HSEQR (H, WR, WI[, ILO][, IHII][, Z][, JOB][, COMPZ][, INFO])
HSEQR (H, W[, ILO][, IHII][, Z][, JOB][, COMPZ][, INFO])
HSEIN (H, WR, WI[, SELECT][, VL][, VR][, IFAILL][, IFAILR][, INITV][, EIGSRC][, M][, INFO])
HSEIN (H, W[, SELECT][, VL][, VR][, IFAILL][, IFAILR][, INITV][, EIGSRC][, M][, INFO])
TREVC (T[, HOWMNY][, SELECT][, VL][, VR][, M][, INFO])
TRSNA (T[, S][, SEP][, VL][, VR][, SELECT][, M][, INFO])
TREXC (T, IFST, ILST[, Q][, INFO])
TRSEN (T, SELECT[, WR][, WI][, M][, S][, SEP][, Q][, INFO])
TRSEN (T, SELECT[, W][, M][, S][, SEP][, Q][, INFO])
TRSYL (A, B, C, SCALE[, TRANA][, TRANB][, ISGN][, INFO])
GGHRD (A, B[, ILO][, IHII][, Q][, Z][, COMPQ][, COMPZ][, INFO])
GGBAL (A, B[, ILO][, IHII][, LSCALE][, RSCALE][, JOB][, INFO])
GGBAK (V[, ILO][, IHII][, LSCALE][, RSCALE][, JOB][, INFO])
HGEQZ (H, T[, ILO][, IHII][, ALPHAR][, ALPHAI][, BETA][, Q][, Z][, JOB][, COMPO][, COMPZ][, INFO])
HGEQZ (H, T[, ILO][, IHII][, ALPHA][, BETA][, Q][, Z][, JOB][, COMPO][, COMPZ][, INFO])
TGEVC (S, P[, HOWMNY][, SELECT][, VL][, VR][, M][, INFO])
TGEXC (A, B[, IFST][, ILST][, Z][, Q][, INFO])
TGSEN (A, B, SELECT[, ALPHAR][, ALPHAI][, BETA][, IJOB][, Q][, Z][, PL][, PR][, DIF][, M][, INFO])
TGSEN (A, B, SELECT[, ALPHA][, BETA][, IJOB][, Q][, Z][, PL][, PR][, DIF][, M][, INFO])
TGSYL (A, B, C, D, E, F[, IJOB][, TRANS][, SCALE][, DIF][, INFO])
TGSNA (A, B[, S][, DIF][, VL][, VR][, SELECT][, M][, INFO])
GGSVP (A, B, TOLA, TOLB[, K][, L][, U][, V][, Q][, INFO])
TGSJA (A, B, TOLA, TOLB, K, L[, U][, V][, Q][, JOBU][, JOBV][, JOBQ][, ALPHA][, BETA][, NCYCLE][, INFO])

```

Interfaces of New Functionality

```
GETRF(A[,IPIV][,INFO])
!  Interface netlib95 exists, parameters:
!  netlib: (A,IPIV,RCOND,NORM,INFO)
!  Different number for parameter, netlib: 5, mkl: 3
!  Different order for parameter INFO, netlib: 5, mkl: 3
!  Absent mkl parameter: NORM
!  Absent mkl parameter: RCOND
```

```
GBTRF(A[,KL][,M][,IPIV][,INFO])
!  Interface netlib95 exists, parameters:
!  netlib: (A,K,M,IPIV,RCOND,NORM,INFO)
!  Different number for parameter, netlib: 7, mkl: 5
!  Different order for parameter INFO, netlib: 7, mkl: 5
!  Absent mkl parameter: NORM
!  Replace parameter name: netlib: K: mkl: KL
!  Absent mkl parameter: RCOND
```

```
POTRF(A[,UPLO][,INFO])
!  Interface netlib95 exists, parameters:
!  netlib: (A,UPLO,RCOND,NORM,INFO)
!  Different number for parameter, netlib: 5, mkl: 3
!  Different order for parameter INFO, netlib: 5, mkl: 3
!  Absent mkl parameter: NORM
!  Absent mkl parameter: RCOND
```

FFTW Interface to Intel® Math Kernel Library



Intel® Math Kernel Library (Intel® MKL) offers FFTW2 and FFTW3 interfaces to Intel MKL Fast Fourier Transform and Trigonometric Transform functionality. The purpose of these interfaces is to enable applications using FFTW (www.fftw.org) to gain performance with Intel MKL without changing the program source code.

Both FFTW2 and FFTW3 interfaces are provided in open source as FFTW wrappers to Intel MKL. For ease of use, FFTW3 interface is also integrated in Intel MKL.

Notational Conventions

This appendix typically employs path notations for Windows* OS.

FFTW2 Interface to Intel® Math Kernel Library

This section describes a collection of C and Fortran wrappers providing FFTW 2.x interface to Intel MKL. The wrappers translate calls to FFTW 2.x functions into the calls of the Intel MKL Fast Fourier Transform interface (FFT interface).

Note that Intel MKL FFT interface operates on both single- and double-precision floating-point data types.

Because of differences between FFTW and Intel MKL FFT functionalities, there are restrictions on using wrappers instead of the FFTW functions. Some FFTW functions have empty wrappers. However, many typical FFTs can be computed using these wrappers.

Refer to [chapter 11 "Fourier Transform Functions"](#), for better understanding the effects from the use of the wrappers.

Wrappers Reference

The section provides a brief reference for the FFTW 2.x C interface. For details please refer to the original FFTW 2.x documentation available at www.fftw.org.

Each FFTW function has its own wrapper. Some of them, which are *not* expressly listed in this section, are empty and do nothing, but they are provided to avoid link errors and satisfy the function calls.

See Also

[Limitations of the FFTW2 Interface to Intel MKL](#)

One-dimensional Complex-to-complex FFTs

The following functions compute a one-dimensional complex-to-complex Fast Fourier transform.

```
fftw_plan fftw_create_plan(int n, fftw_direction dir, int flags);
fftw_plan fftw_create_plan_specific(int n, fftw_direction dir, int flags, fftw_complex *in, int istride, fftw_complex *out, int ostride);

void fftw(fftw_plan plan, int howmany, fftw_complex *in, int istride, int idist, fftw_complex *out, int ostride, int odist);

void fftw_one(fftw_plan plan, fftw_complex *in , fftw_complex *out);

void fftw_destroy_plan(fftw_plan plan);
```

Multi-dimensional Complex-to-complex FFTs

The following functions compute a multi-dimensional complex-to-complex Fast Fourier transform.

```
fftwnd_plan fftwnd_create_plan(int rank, const int *n, fftw_direction dir, int flags);
fftwnd_plan fftw2d_create_plan(int nx, int ny, fftw_direction dir, int flags);
fftwnd_plan fftw3d_create_plan(int nx, int ny, int nz, fftw_direction dir, int flags);
fftwnd_plan fftwnd_create_plan_specific(int rank, const int *n, fftw_direction dir, int flags,
                                         fftw_complex *in, int istride, fftw_complex *out, int ostride);
fftwnd_plan fftw2d_create_plan_specific(int nx, int ny, fftw_direction dir, int flags,
                                         fftw_complex *in, int istride, fftw_complex *out, int ostride);
fftwnd_plan fftw3d_create_plan_specific(int nx, int ny, int nz, fftw_direction dir, int flags,
                                         fftw_complex *in, int istride, fftw_complex *out, int ostride);

void fftwnd(fftwnd_plan plan, int howmany, fftw_complex *in, int istride, int idist,
            fftw_complex *out, int ostride, int odist);
void fftwnd_one(fftwnd_plan plan, fftw_complex *in, fftw_complex *out);
void fftwnd_destroy_plan(fftwnd_plan plan);
```

One-dimensional Real-to-half-complex/Half-complex-to-real FFTs

Half-complex representation of a conjugate-even symmetric vector of size N in a real array of the same size N consists of $N/2+1$ real parts of the elements of the vector followed by non-zero imaginary parts in the reverse order. Because the Intel MKL FFT interface does not currently support this representation, all wrappers of this kind are empty and do nothing.

Nevertheless, you can perform one-dimensional real-to-complex and complex-to-real transforms using rfftwnd functions with $rank=1$.

See Also

[Multi-dimensional Real-to-complex/Complex-to-real FFTs](#)

Multi-dimensional Real-to-complex/Complex-to-real FFTs

The following functions compute multi-dimensional real-to-complex and complex-to-real Fast Fourier transforms.

```
rfftwnd_plan rfftwnd_create_plan(int rank, const int *n, fftw_direction dir, int flags);
rfftwnd_plan rfftw2d_create_plan(int nx, int ny, fftw_direction dir, int flags);
rfftwnd_plan rfftw3d_create_plan(int nx, int ny, int nz, fftw_direction dir, int flags);

rfftwnd_plan rfftwnd_create_plan_specific(int rank, const int *n, fftw_direction dir, int flags,
                                           fftw_real *in, int istride, fftw_real *out, int ostride);
rfftwnd_plan rfftw2d_create_plan_specific(int nx, int ny, fftw_direction dir, int flags,
                                           fftw_real *in, int istride, fftw_real *out, int ostride);
rfftwnd_plan rfftw3d_create_plan_specific(int nx, int ny, int nz, fftw_direction dir, int flags,
                                           fftw_real *in, int istride, fftw_real *out, int ostride);

void rfftwnd_real_to_complex(rfftwnd_plan plan, int howmany, fftw_real *in, int istride, int idist,
                             fftw_complex *out, int ostride, int odist);
void rfftwnd_complex_to_real(rfftwnd_plan plan, int howmany, fftw_complex *in, int istride, int idist,
                             fftw_real *out, int ostride, int odist);
void rfftwnd_one_real_to_complex(rfftwnd_plan plan, fftw_real *in, fftw_complex *out);
```

```
void rfftwnd_one_complex_to_real(rfftwnd_plan plan, fftw_complex *in, fftw_real *out);
void rfftwnd_destroy_plan(rfftwnd_plan plan);
```

Multi-threaded FFTW

This section discusses multi-threaded FFTW wrappers only. MPI FFTW wrappers, available only with Intel MKL for the Linux* and Windows* operating systems, are described in [section "MPI FFTW Wrappers"](#).

Unlike the original FFTW interface, every computational function in the FFTW2 interface to Intel MKL provides multithreaded computation by default, with the number of threads defined by the number of processors available on the system (see [section "Managing Performance and Memory" in the Intel MKL User's Guide](#)). To limit the number of threads that use the FFTW interface, call the threaded FFTW computational functions:

```
void fftw_threads(int nthreads, fftw_plan plan, int howmany, fftw_complex *in, int
  istride, int idist, fftw_complex *out, int ostride, int odist);
void fftw_threads_one(int nthreads, rfftwnd_plan plan, fftw_complex *in, fftw_complex
  *out);
...
void rfftwnd_threads_real_to_complex( int nthreads, rfftwnd_plan plan, int
  howmany, fftw_real *in, int istride, int idist, fftw_complex *out, int ostride, int
  odist);
```

Compared to its non-threaded counterpart, every threaded computational function has `threads_` as the second part of its name and additional first parameter `nthreads`. Set the `nthreads` parameter to the thread limit to ensure that the computation requires at most that number of threads.

FFTW Support Functions

The FFTW wrappers provide memory allocation functions to be used with FFTW:

```
void* fftw_malloc(size_t n);
void fftw_free(void* x);
```

The `fftw_malloc` wrapper aligns the memory on a 16-byte boundary.

If `fftw_malloc` fails to allocate memory, it aborts the application. To override this behavior, set a global variable `fftw_malloc_hook` and optionally the complementary variable `fftw_free_hook`:

```
void *(*fftw_malloc_hook) (size_t n);
void (*fftw_free_hook) (void *p);
```

The wrappers use the function `fftw_die` to abort the application in cases when a caller cannot be informed of an error otherwise (for example, in computational functions that return `void`). To override this behavior, set a global variable `fftw_die_hook`:

```
void (*fftw_die_hook)(const char *error_string);
void fftw_die(const char *s);
```

Calling Wrappers from Fortran

The FFTW2 wrappers to Intel MKL provide the following subroutines for calling from Fortran:

```
call fftw_f77_create_plan(plan, n, dir, flags)
call fftw_f77(plan, howmany, in, istride, idist, out, ostride, odist)
call fftw_f77_one(plan, in, out)
call fftw_f77_threads(nthreads, plan, howmany, in, istride, idist, out, ostride,
odist)
call fftw_f77_threads_one(nthreads, plan, in, out)
```

```

call fftw_f77_destroy_plan(plan)

call fftwnd_f77_create_plan(plan, rank, n, dir, flags)
call fftw2d_f77_create_plan(plan, nx, ny, dir, flags)
call fftw3d_f77_create_plan(plan, nx, ny, nz, dir, flags)
call fftwnd_f77(plan, howmany, in, istride, idist, out, ostride, odist)
call fftwnd_f77_one(plan, in, out)
call fftwnd_f77_threads(nthreads, plan, howmany, in, istride, idist, out, ostride,
odist)
call fftwnd_f77_threads_one(nthreads, plan, in, out)
call fftwnd_f77_destroy_plan(plan)

call rfftw_f77_create_plan(plan, n, dir, flags)
call rfftw_f77(plan, howmany, in, istride, idist, out, ostride, odist)
call rfftw_f77_one(plan, in, out)
call rfftw_f77_threads(nthreads, plan, howmany, in, istride, idist, out, ostride,
odist)
call rfftw_f77_threads_one(nthreads, plan, in, out)
call rfftw_f77_destroy_plan(plan)

call rfftwnd_f77_create_plan(plan, rank, n, dir, flags)
call rfftw2d_f77_create_plan(plan, nx, ny, dir, flags)
call rfftw3d_f77_create_plan(plan, nx, ny, nz, dir, flags)
call rfftwnd_f77_complex_to_real(plan, howmany, in, istride, idist, out, ostride,
odist)
call rfftwnd_f77_one_complex_to_real(plan, in, out)
call rfftwnd_f77_real_to_complex(plan, howmany, in, istride, idist, out, ostride,
odist)
call rfftwnd_f77_one_real_to_complex(plan, in, out)
call rfftwnd_f77_threads_complex_to_real(nthreads, plan, howmany, in, istride, idist,
out, ostride, odist)
call rfftwnd_f77_threads_one_complex_to_real(nthreads, plan, in, out)
call rfftwnd_f77_threads_real_to_complex(nthreads, plan, howmany, in, istride, idist,
out, ostride, odist)
call rfftwnd_f77_threads_one_real_to_complex(nthreads, plan, in, out)
call rfftwnd_f77_destroy_plan(plan)

call fftw_f77_threads_init(info)

```

The FFTW Fortran functions are wrappers to FFTW C functions. So, their functionality and limitations are the same as of the corresponding C wrappers.

See also these resources:

[Wrappers Reference](#)

for a description of the FFTW2 C wrappers to Intel MKL.

www.fftw.org

for the original FFTW 2.x documentation.

[Limitations of the FFTW2 Interface to Intel MKL](#) for limitations of the wrappers.

Limitations of the FFTW2 Interface to Intel MKL

The FFTW2 wrappers implement the functionality of only those FFTW functions that Intel MKL can reasonably support. Other functions are provided as no-operation functions, whose only purpose is to satisfy link-time symbol resolution. Specifically, no-operation functions include:

- Real-to-half-complex and respective backward transforms
- Print plan functions
- Functions for importing/exporting/forgetting wisdom
- Most of the FFTW functions not covered by the original FFTW2 documentation

Because the Intel MKL implementation of FFTW2 wrappers does not use plan and plan node structures declared in `fftw.h`, the behavior of an application that relies on the internals of the plan structures defined in that header file is undefined.

FFTW2 wrappers define plan as a set of attributes, such as strides, used to commit the Intel MKL FFT descriptor structure. If an FFTW2 computational function is called with attributes different from those recorded in the plan, the function attempts to adjust the attributes of the plan and recommit the descriptor. So, repeated calls of a computational function with the same plan but different strides, distances, and other parameters may be performance inefficient.

Plan creation functions disregard most planner flags passed through the `flags` parameter. These functions take into account only the following values of `flags`:

- `FFTW_IN_PLACE`

If this value of `flags` is supplied, the plan is marked so that computational functions using that plan ignore the parameters related to output (`out`, `ostride`, and `odist`). Unlike the original FFTW interface, the wrappers never use the `out` parameter as a scratch space for in-place transforms.

- `FFTW_THREADSAFE`

If this value of `flags` is supplied, the plan is marked read-only. An attempt to change attributes of a read-only plan aborts the application.

FFTW wrappers are generally not thread safe. Therefore, do not use the same plan in parallel user threads simultaneously.

Installation

Wrappers are delivered as source code, which you must compile to build the wrapper library. Then you can substitute the wrapper and Intel MKL libraries for the FFTW library. The source code for the wrappers, makefiles, and files with lists of wrappers are located in the `.\interfaces\fftw2xc` or `.\interfaces\fftw2xf` subdirectory in the Intel MKL directory for C and Fortran wrappers, respectively.

Creating the Wrapper Library

Two header files are used to compile the C wrapper library: `fftw2_mkl.h` and `fftw.h`. The `fftw2_mkl.h` file is located in the `.\interfaces\fftw2xc\wrappers` subdirectory in the Intel MKL directory.

Three header files are used to compile the Fortran wrapper library: `fftw2_mkl.h`, `fftw2_f77_mkl.h`, and `fftw.h`. The `fftw2_mkl.h` and `fftw2_f77_mkl.h` files are located in the `.\interfaces\fftw2xf\wrappers` subdirectory in the Intel MKL directory.

The file `fftw.h`, used to compile libraries for both interfaces and located in the `.\include\fftw` subdirectory in the Intel MKL directory, slightly differs from the original FFTW (www.fftw.org) header file `fftw.h`.

The source code for the wrappers, makefiles, and files with lists of functions are located in the `.\interfaces\fftw2xc` or `.\interfaces\fftw2xf` subdirectory in the Intel MKL directory for C or Fortran wrappers, respectively.

A wrapper library contains C or Fortran wrappers for complex and real transforms in a serial and multi-threaded mode for double- or single-precision floating-point data types. A makefile parameter manages the data type.

Parameters of a makefile also specify the platform (required), compiler, and data precision. The makefile comment heading provides the exact description of these parameters.

Because a C compiler builds the Fortran wrapper library, function names in the wrapper library and Fortran object module may be different. The file `fftw2_f77_mkl.h` in the `.\interfaces\fftw2xf\source` subdirectory in the Intel MKL directory defines function names according to the names in the Fortran module. If a required name is missing in the file, you can modify the file to add the name before building the library.

To build the library, run the `make` command on Linux* OS and OS X* or the `nmake` command on Windows* OS with appropriate parameters.

For example, on Linux OS the command

```
make libintel64
```

builds a double-precision wrapper library for Intel® 64 architecture based applications using the Intel® C++ Compiler or the Intel® Fortran Compiler (Compilers and data precision are chosen by default.)

Each makefile creates the library in the directory with Intel MKL libraries corresponding to the platform used. For example, `./lib/ia32` (on Linux OS and OS X) or `.\lib\ia32` (on Windows* OS).

In the names of a wrapper library, the suffix corresponds to the compiler used and the letter preceding the underscore is "f" for the Fortran and "c" for the C programming language.

For example,

```
fftw2xf_intel.lib (on Windows OS); libfftw2xf_intel.a (on Linux OS and OS X);
fftw2xc_intel.lib (on Windows OS); libfftw2xc_intel.a (on Linux OS and OS X);
fftw2xc_ms.lib (on Windows OS); libfftw2xc_gnu.a (on Linux OS and OS X).
```

Application Assembling

Use the necessary original FFTW (www.fftw.org) header files without any modifications. Use the created wrapper library and the Intel MKL library instead of the FFTW library.

Running Examples

Intel MKL provides examples to demonstrate how to use the MPI FFTW wrapper library. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw2xc` or `.\examples\fftw2xf` subdirectory in the Intel MKL directory for C or Fortran, respectively. To build examples, several additional files are needed: `fftw.h`, `fftw_threads.h`, `rfftw.h`, `rfftw_threads.h`, and `fftw_f77.i`. These files are distributed with permission from FFTW and are available in `.\include\fftw`. The original files can also be found in FFTW 2.1.5 at <http://www.fftw.org/download.html>.

An example makefile uses the `function` parameter in addition to the parameters of the corresponding wrapper library makefile (see [Creating a Wrapper Library](#)). The makefile comment heading provides the exact description of these parameters.

An example makefile normally invokes examples. However, if the appropriate wrapper library is not yet created, the makefile first builds the library the same way as the wrapper library makefile does and then proceeds to examples.

If the parameter `function=<example_name>` is defined, only the specified example runs. Otherwise, all examples from the appropriate subdirectory run. The subdirectory `_results` is created, and the results are stored there in the `<example_name>.res` files.

MPI FFTW Wrappers

MPICH wrappers for FFTW 2 are available only with Intel® MKL for the Linux* and Windows* operating systems.

MPI FFTW Wrappers Reference

The section provides a reference for MPI FFTW C interface.

Complex MPI FFTW

Complex One-dimensional MPI FFTW Transforms

```
fftw_mpi_plan fftw_mpi_create_plan(MPI_Comm comm, int n, fftw_direction dir, int flags);
void fftw_mpi(fftw_mpi_plan p, int n_fields, fftw_complex *local_data, fftw_complex *work);
void fftw_mpi_local_sizes(fftw_mpi_plan p, int *local_n, int *local_start, int *local_n_after_transform, int *local_start_after_transform, int *total_local_size);
void fftw_mpi_destroy_plan(fftw_mpi_plan plan);
```

Argument restrictions:

- Supported values of *flags* are FFTW_ESTIMATE, FFTW_MEASURE, FFTW_SCRAMBLED_INPUT and FFTW_SCRAMBLED_OUTPUT. The same algorithm corresponds to all these values of the *flags* parameter. If any other *flags* value is supplied, the wrapper library reports an error '*CDFT error in wrapper: unknown flags*'.
- The only supported value of *n_fields* is 1.

Complex Multi-dimensional MPI FFTW Transforms

```
fftwnd_mpi_plan fftw2d_mpi_create_plan(MPI_Comm comm, int nx, int ny, fftw_direction dir, int flags);
fftwnd_mpi_plan fftw3d_mpi_create_plan(MPI_Comm comm, int nx, int ny, int nz, fftw_direction dir, int flags);
fftwnd_mpi_plan fftwnd_mpi_create_plan(MPI_Comm comm, int dim, int *n, fftw_direction dir, int flags);

void fftwnd_mpi(fftwnd_mpi_plan p, int n_fields, fftw_complex *local_data, fftw_complex *work, fftwnd_mpi_output_order output_order);
void fftwnd_mpi_local_sizes(fftwnd_mpi_plan p, int *local_nx, int *local_x_start, int *local_ny_after_transpose, int *local_y_start_after_transpose, int *total_local_size);
void fftwnd_mpi_destroy_plan(fftwnd_mpi_plan plan);
```

Argument restrictions:

- Supported values of *flags* are FFTW_ESTIMATE and FFTW_MEASURE. If any other value of *flags* is supplied, the wrapper library reports an error '*CDFT error in wrapper: unknown flags*'.
- The only supported value of *n_fields* is 1.

Real MPI FFTW

Real-to-Complex MPI FFTW Transforms

```
rfftwnd_mpi_plan rfftw2d_mpi_create_plan(MPI_Comm comm, int nx, int ny, fftw_direction dir, int flags);
rfftwnd_mpi_plan rfftw3d_mpi_create_plan(MPI_Comm comm, int nx, int ny, int nz, fftw_direction dir, int flags);
```

```
rfftwnd_mpi_plan rfftwnd_mpi_create_plan(MPI_Comm comm, int dim, int *n, fftw_direction dir, int flags);

void rfftwnd_mpi(rfftwnd_mpi_plan p, int n_fields, fftw_real *local_data, fftw_real *work, fftwnd_mpi_output_order output_order);

void rfftwnd_mpi_local_sizes(rfftwnd_mpi_plan p, int *local_nx, int *local_x_start, int *local_ny_after_transpose, int *local_y_start_after_transpose, int *total_local_size);

void rfftwnd_mpi_destroy_plan(rfftwnd_mpi_plan plan);
```

Argument restrictions:

- Supported values of *flags* are FFTW_ESTIMATE and FFTW_MEASURE. If any other value of *flags* is supplied, the wrapper library reports an error 'CDFT error in wrapper: unknown flags'.
- The only supported value of *n_fields* is 1.

-
- Function *rfftwnd_mpi_create_plan* can be used for both one-dimensional and multi-dimensional transforms.
 - Both values of the *output_order* parameter are supported: FFTW_NORMAL_ORDER and FFTW_TRANSPOSED_ORDER.
-

Creating MPI FFTW Wrapper Library

The source code for the wrappers, makefiles, and files with lists of wrappers are located in the .\interfaces\fftw2x_cdft subdirectory in the Intel MKL directory.

A wrapper library contains C wrappers for Complex One-dimensional MPI FFTW Transforms and Complex Multi-dimensional MPI FFTW Transforms. The library also contains empty C wrappers for Real Multi-dimensional MPI FFTW Transforms. For details, see [MPI FFTW Wrappers Reference](#).

Parameters of a makefile specify the platform (required), compiler, and data precision. Specifying the platform is required. The makefile comment heading provides the exact description of these parameters.

To build the library, run the `make` command on Linux* OS and OS X* or the `nmake` command on Windows* OS with appropriate parameters.

For example, on Linux OS the command

```
make libintel64
```

builds a double-precision wrapper library for Intel® 64 architecture based applications using Intel MPI and the Intel® C++ Compiler (compilers and data precision are chosen by default.).

A makefile creates the wrapper library in the directory with the Intel MKL libraries corresponding to the used platform. For example, `./lib/ia32` (on Linux OS) or `.\lib\ia32` (on Windows* OS).

In the wrapper library names, the suffix corresponds to the used data precision. For example,

`fftw2x_cdft_SINGLE.lib` on Windows OS;

`libfftw2x_cdft_DOUBLE.a` on Linux OS.

Application Assembling with MPI FFTW Wrapper Library

Use the necessary original FFTW (www.fftw.org) header files without any modifications. Use the created MPI FFTW wrapper library and the Intel MKL library instead of the FFTW library.

Running Examples

There are some examples that demonstrate how to use the MPI FFTW wrapper library for FFTW2. The source C code for the examples, makefiles used to run them, and files with lists of examples are located in the .\examples\fftw2x_cdft subdirectory in the Intel MKL directory. To build examples, one additional file `fftw_mpi.h` is needed. This file is distributed with permission from FFTW and is available in .\include\fftw. The original file can also be found in FFTW 2.1.5 at <http://www.fftw.org/download.html>.

Parameters for the example makefiles are described in the makefile comment headings and are similar to the parameters of the wrapper library makefiles (see [Creating MPI FFTW Wrapper Library](#)).

The table below lists examples available in the .\examples\fftw2x_cdft\source subdirectory.

Examples of MPI FFTW Wrappers

Source file for the example	Description
<code>wrappers_c1d.c</code>	One-dimensional Complex MPI FFTW transform, using <code>plan = fftw_mpi_create_plan(...)</code>
<code>wrappers_c2d.c</code>	Two-dimensional Complex MPI FFTW transform, using <code>plan = fftw2d_mpi_create_plan(...)</code>
<code>wrappers_c3d.c</code>	Three-dimensional Complex MPI FFTW transform, using <code>plan = fftw3d_mpi_create_plan(...)</code>
<code>wrappers_c4d.c</code>	Four-dimensional Complex MPI FFTW transform, using <code>plan = fftwnd_mpi_create_plan(...)</code>
<code>wrappers_r1d.c</code>	One-dimensional Real MPI FFTW transform, using <code>plan = rfftw_mpi_create_plan(...)</code>
<code>wrappers_r2d.c</code>	Two-dimensional Real MPI FFTW transform, using <code>plan = rfftw2d_mpi_create_plan(...)</code>
<code>wrappers_r3d.c</code>	Three-dimensional Real MPI FFTW transform, using <code>plan = rfftw3d_mpi_create_plan(...)</code>
<code>wrappers_r4d.c</code>	Four-dimensional Real MPI FFTW transform, using <code>plan = rfftwnd_mpi_create_plan(...)</code>

FFTW3 Interface to Intel® Math Kernel Library

This section describes a collection of FFTW3 wrappers to Intel MKL. The wrappers translate calls of FFTW3 functions to the calls of the Intel MKL Fourier transform (FFT) or Trigonometric Transform (TT) functions. The purpose of FFTW3 wrappers is to enable developers whose programs currently use the FFTW3 library to gain performance with the Intel MKL Fourier transforms without changing the program source code.

The FFTW3 wrappers provide a limited functionality compared to the original FFTW 3.x library, because of differences between FFTW and Intel MKL FFT and TT functionality. This section describes limitations of the FFTW3 wrappers and hints for their usage. Nevertheless, many typical FFT tasks can be performed using the FFTW3 wrappers to Intel MKL.

The FFTW3 wrappers are integrated in Intel MKL. The only change required to use Intel MKL through the FFTW3 wrappers is to link your application using FFTW3 against Intel MKL.

A reference implementation of the FFTW3 wrappers is also provided in open source. You can find it in the `interfaces` directory of the Intel MKL distribution. You can use the reference implementation to create your own wrapper library (see [Building Your Own Wrapper Library](#))

See also these resources:

Intel MKL Release Notes

for the version of the FFTW3 library supported by the wrappers.

www.fftw.org	for a description of the FFTW interface.
Fourier Transform Functions	for a description of the Intel MKL FFT interface.
Trigonometric Transform Routines	for a description of Intel MKL TT interface.

Using FFTW3 Wrappers

The FFTW3 wrappers are a set of functions and data structures depending on one another. The wrappers are not designed to provide the interface on a function-per-function basis. Some FFTW3 wrapper functions are empty and do nothing, but they are present to avoid link errors and satisfy function calls.

This manual does not list the declarations of the functions that the FFTW3 wrappers provide (you can find the declarations in the `fftw3.h` header file). Instead, this section comments particular limitations of the wrappers and provides usage hints:

- The FFTW3 wrappers do not support long double precision because Intel MKL FFT functions operate only on single- and double-precision floating-point data types (`float` and `double`, respectively). Therefore the functions with prefix `fftwl_`, supporting the `long double` data type, are not provided.
- The wrappers provide equivalent implementation for double- and single-precision functions (those with prefixes `fftw_` and `fftwf_`, respectively). So, all these comments equally apply to the double- and single-precision functions and will refer to functions with prefix `fftw_`, that is, double-precision functions, for brevity.
- The FFTW3 interface that the wrappers provide is defined in the `fftw3.h` and `fftw3.f` header files. These files are borrowed from the FFTW3.x package and distributed within Intel MKL with permission. Additionally, the `fftw3_mkl.h`, `fftw3_mkl.f`, and `fftw3_mkl_f77.h` header files define supporting structures and supplementary constants and macros, as well as expose Fortran interface in C.
- Actual functionality of the plan creation wrappers is implemented in guru64 set of functions. Basic interface, advanced interface, and guru interface plan creation functions call the guru64 interface functions. So, all types of the FFTW3 plan creation interface in the wrappers are functional.
- Plan creation functions may return a `NULL` plan, indicating that the functionality is not supported. So, please carefully check the result returned by plan creation functions in your application. In particular, the following problems return a `NULL` plan:
 - c2r and r2c problems with a split storage of complex data.
 - r2r problems with *kind* values `FFTW_R2HC`, `FFTW_HC2R`, and `FFTW_DHT`. The only supported r2r kinds are even/odd DFTs (sine/cosine transforms).
 - Multidimensional r2r transforms.
 - Transforms of multidimensional vectors. That is, the only supported values for parameter `howmany_rank` in guru and guru64 plan creation functions are 0 and 1.
 - Multidimensional transforms with `rank > MKL_MAXRANK`.
- The `MKL_RODFT00` value of the *kind* parameter is introduced by the FFTW3 wrappers. For better performance, you are strongly encouraged to use this value rather than `FFTW_RODFT00`. To use this *kind* value, provide an extra first element equal to 0.0 for the input/output vectors. Consider the following example:

```
plan1 = fftw_plan_r2r_1d(n, in1, out1, FFTW_RODFT00, FFTW_ESTIMATE);
plan2 = fftw_plan_r2r_1d(n, in2, out2, MKL_RODFT00, FFTW_ESTIMATE);
```

Both plans perform the same transform, except that the `in2/out2` arrays have one extra zero element at location 0. For example, if $n=3$, `in1={x,y,z}` and `out1={u,v,w}`, then `in2={0,x,y,z}` and `out2={0,u,v,w}`.

- The *flags* parameter in plan creation functions is always ignored. The same algorithm is used regardless of the value of this parameter. In particular, *flags* values `FFTW_ESTIMATE`, `FFTW_MEASURE`, etc. have no effect.
- For multithreaded plans, use normal sequence of calls to the `fftw_init_threads()` and `fftw_plan_with_nthreads()` functions (refer to FFTW documentation).
- Memory allocation function `fftw_malloc` returns memory aligned at a 16-byte boundary. You must free the memory with `fftw_free`.

- The FFTW3 wrappers to Intel MKL use the 32-bit `int` type in both LP64 and ILP64 interfaces of Intel MKL. Use `guru64` FFTW3 interfaces for 64-bit sizes.
- Fortran wrappers (see [Calling Wrappers from Fortran](#)) use the `INTEGER` type, which is 32-bit in LP64 interfaces and 64-bit in ILP64 interfaces.
- The wrappers typically indicate a problem by returning a `NULL` plan. In a few cases, the wrappers may report a descriptive message of the problem detected. By default the reporting is turned off. To turn it on, set variable `fftw3_mkl.verbose` to a non-zero value, for example:

```
#include "fftw3.h"
#include "fftw3_mkl.h"
fftw3_mkl.verbose = 0;
plan = fftw_plan_r2r(...);
```

- The following functions are empty:
 - For saving, loading, and printing plans
 - For saving and loading wisdom
 - For estimating arithmetic cost of the transforms.
- Do not use macro `FFTW_DLL` with the FFTW3 wrappers to Intel MKL.
- Do not use negative stride values. Though FFTW3 wrappers support negative strides in the part of advanced and guru FFTW interface, the underlying implementation does not.

Calling Wrappers from Fortran

Intel MKL also provides Fortran 77 interfaces of the FFTW3 wrappers. The Fortran wrappers are available for all FFTW3 interface functions and are based on C interface of the FFTW3 wrappers. Therefore they have the same functionality and restrictions as the corresponding C interface wrappers.

The Fortran wrappers use the default `INTEGER` type for integer arguments. The default `INTEGER` is 32-bit in Intel MKL LP64 interfaces and 64-bit in ILP64 interfaces. Argument `plan` in a Fortran application must have type `INTEGER*8`.

The wrappers that are double-precision subroutines have prefix `dfftw_`, single-precision subroutines have prefix `sfftw_` and provide an equivalent functionality. Long double subroutines (with prefix `lfftw_`) are not provided.

The Fortran FFTW3 wrappers use the default Intel® Fortran compiler convention for name decoration. If your compiler uses a different convention, or if you are using compiler options affecting the name decoration (such as `/Qlowercase`), you may need to compile the wrappers from sources, as described in section [Building Your Own Wrapper Library](#).

For interoperability with C, the declaration of the Fortran FFTW3 interface is provided in header file `include/fftw/fftw3_mkl_f77.h`.

You can call Fortran wrappers from a FORTRAN 77 or Fortran 90 application, although Intel MKL does not provide a Fortran 90 module for the wrappers. For a detailed description of the FFTW Fortran interface, refer to FFTW3 documentation (www.fftw.org).

The following example illustrates calling the FFTW3 wrappers from Fortran:

```
INTEGER*8 plan
INTEGER N
INCLUDE 'fftw3.f'
COMPLEX*16 IN(*), OUT(*)
!...initialize array IN
CALL DFFT_W_PLAN_DFT_1D(PLAN, N, IN, OUT, -1, FFTW_ESTIMATE)
IF (PLAN .EQ. 0) STOP
CALL DFFT_W_EXECUTE
!...result is in array OUT
```

Building Your Own Wrapper Library

The FFTW3 wrappers to Intel MKL are delivered both integrated in Intel MKL and as source code, which can be compiled to build a standalone wrapper library with exactly the same functionality. Normally you do not need to build the wrappers yourself. However, if your Fortran application is compiled with a compiler that uses a different name decoration than the Intel® Fortran compiler or if you are using compiler options altering the Fortran name decoration, you may need to build the wrappers that use the appropriate name changing convention.

The source code for the wrappers, makefiles, and files with lists of functions are located in the `.\interfaces\fftw3xc` or `.\interfaces\fftw3xf` subdirectory in the Intel MKL directory for C or Fortran wrappers, respectively.

To build the wrappers,

1. Change the current directory to the wrapper directory
2. Run the `make` command on Linux* OS and OS X* or the `nmake` command on Windows* OS with a required target and optionally several parameters.

The target `libia32` or `libintel64` defines the platform architecture, and the other parameters specify the compiler, size of the default integer type, and placement of the resulting wrapper library. You can find a detailed and up-to-date description of the parameters in the makefile.

In the following example, the `make` command is used to build the FFTW3 Fortran wrappers to Intel MKL for use from the GNU g77* Fortran compiler on Linux OS based on Intel® 64 architecture:

```
cd interfaces/fftw3xf
make libintel64 compiler=gnu fname=a_name__ INSTALL_DIR=/my/path
```

This command builds the wrapper library using the GNU gcc compiler, decorates the name with the second underscore, and places the result, named `libfftw3xf_gnu.a`, into the `/my/path` directory. The name of the resulting library is composed of the name of the compiler used and may be changed by an optional parameter `INSTALL_LIBNAME`.

Building an Application

Normally, the only change needed to build your application with FFTW3 wrappers replacing original FFTW library is to add Intel MKL at the link stage (see section *"Linking Your Application with Intel® Math Kernel Library" in the Intel MKL User's Guide*).

If you recompile your application, add subdirectory `include\fftw` to the search path for header files to avoid FFTW3 version conflicts.

Sometimes, you may have to modify your application according to the following recommendations:

- The application requires


```
#include "fftw3.h" ,
```

 which it probably already includes.
- The application does not require


```
#include "mkl_dfti.h" .
```
- The application does not require


```
#include "fftw3_mkl.h" .
```

It is required only in case you want to use the `MKL_RODFT00` constant.

- If the application does not check whether a `NULL` plan is returned by plan creation functions, this check must be added, because the FFTW3 to Intel MKL wrappers do not provide 100% of FFTW3 functionality.

Running Examples

There are some examples that demonstrate how to use the wrapper library. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the .\examples\fftw3xc or .\examples\fftw3xf subdirectory in the Intel MKL directory for C or Fortran, respectively. To build Fortran examples, one additional file fftw3.f is needed. This file is distributed with permission from FFTW and is available in the .\include\fftw subdirectory of the Intel MKL directory. The original file can also be found in FFTW 3.2 at <http://www.fftw.org/download.html>.

Parameters of the example makefiles are similar to the parameters of the wrapper library makefiles. Example makefiles normally build and invoke the examples. If the parameter `function=<example_name>` is defined, then only the specified example will run. Otherwise, all examples will be executed. Results of running the examples are saved in subdirectory ._results in files with extension .res.

For detailed information about options for the example makefile, refer to the makefile.

MPI FFTW Wrappers

This section describes a collection of MPI FFTW wrappers to Intel® MKL.

MPI FFTW wrappers are available only with Intel MKL for the Linux* and Windows* operating systems.

These wrappers translate calls of MPI FFTW functions to the calls of the Intel MKL cluster Fourier transform (CFFT) functions. The purpose of the wrappers is to enable users of MPI FFTW functions improve performance of the applications without changing the program source code.

Although the MPI FFTW wrappers provide less functionality than the original FFTW3 because of differences between MPI FFTW and Intel MKL CFFT, the wrappers cover many typical CFFT use cases.

The MPI FFTW wrappers are provided as source code. To use the wrappers, you need to build your own wrapper library (see [Building Your Own Wrapper Library](#)).

See also these resources:

[Intel MKL Release Notes](#) for the version of the FFTW3 library supported by the wrappers.

www.fftw.org for a description of the MPI FFTW interface.

[Cluster FFT Functions](#) for a description of the Intel MKL CFFT interface.

Building Your Own Wrapper Library

The MPI FFTW wrappers for FFTW3 are delivered as source code, which can be compiled to build a wrapper library.

The source code for the wrappers, makefiles, and files with lists of functions are located in subdirectory .\interfaces\fftw3x_cdft in the Intel MKL directory.

To build the wrappers,

1. Change the current directory to the wrapper directory
2. Run the `make` command on Linux* OS or the `nmake` command on Windows* OS with a required target and optionally several parameters.

The target `libia32` or `libintel64` defines the platform architecture, and the other parameters specify the compiler, size of the default `INTEGER` type, as well as the name and placement of the resulting wrapper library. You can find a detailed and up-to-date description of the parameters in the makefile.

In the following example, the `make` command is used to build the MPI FFTW wrappers to Intel MKL for use from the GNU C compiler on Linux OS based on Intel® 64 architecture:

```
cd interfaces\fftw3x_cdft
make libintel64 compiler=gnu mpi=openmpi INSTALL_DIR=/my/path
```

This command builds the wrapper library using the GNU gcc compiler so that the final executable can use Open MPI, and places the result, named `libfftw3x_cdft_DOUBLE.a`, into directory `/my/path`.

Building an Application

Normally, the only change needed to build your application with MPI FFTW wrappers replacing original FFTW3 library is to add Intel MKL and the wrapper library at the link stage (see section "*Linking Your Application with Intel® Math Kernel Library*" in the *Intel MKL User's Guide*).

When you are recompiling your application, add subdirectory `include\fftw` to the search path for header files to avoid FFTW3 version conflicts.

Running Examples

There are some examples that demonstrate how to use the MPI FFTW wrapper library for FFTW3. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\\examples\\fftw3x_cdft` subdirectory in the Intel MKL directory.

Parameters of the example makefiles are similar to the parameters of the wrapper library makefiles. Example makefiles normally build and invoke the examples. Results of running the examples are saved in subdirectory `._results` in files with extension `.res`.

For detailed information about options for the example makefile, refer to the makefile.

See Also

[Building Your Own Wrapper Library](#)

Code Examples

This appendix presents code examples of using some Intel MKL routines and functions. You can find here example code written in both Fortran and C.

Please refer to respective chapters in the manual for detailed descriptions of function parameters and operation.

BLAS Code Examples

Example. Using BLAS Level 1 Function

The following example illustrates a call to the BLAS Level 1 function `sdot`. This function performs a vector-vector operation of computing a scalar product of two single-precision real vectors x and y .

Parameters

<code>n</code>	Specifies the number of elements in vectors x and y .
<code>incx</code>	Specifies the increment for the elements of x .
<code>incy</code>	Specifies the increment for the elements of y .

```
program dot_main
real x(10), y(10), sdot, res
integer n, incx, incy, i
external sdot
n = 5
incx = 2
incy = 1
do i = 1, 10
  x(i) = 2.0e0
  y(i) = 1.0e0
end do
res = sdot (n, x, incx, y, incy)
print*, `SDOT = `, res
end
```

As a result of this program execution, the following line is printed:

`SDOT = 10.000`

Example. Using BLAS Level 1 Routine

The following example illustrates a call to the BLAS Level 1 routine `scopy`. This routine performs a vector-vector operation of copying a single-precision real vector x to a vector y .

Parameters

<code>n</code>	Specifies the number of elements in vectors x and y .
<code>incx</code>	Specifies the increment for the elements of x .

Parameters

incy Specifies the increment for the elements of *y*.

```

program copy_main
real x(10), y(10)
integer n, incx, incy, i
n = 3
incx = 3
incy = 1
do i = 1, 10
    x(i) = i
end do
call scopy (n, x, incx, y, incy)
print*, `Y = `, (y(i), i = 1, n)
end

```

As a result of this program execution, the following line is printed:

Y = 1.00000 4.00000 7.00000

Example. Using BLAS Level 2 Routine

The following example illustrates a call to the BLAS Level 2 routine `sger`. This routine performs a matrix-vector operation

a := *alpha***x***y*' + *a*.

Parameters

α	Specifies a scalar α .
x	m -element vector.
y	n -element vector.
A	m -by- n matrix.

```

program ger_main
real a(5,3), x(10), y(10), alpha
integer m, n, incx, incy, i, j, lda
m = 2
n = 3
lda = 5
incx = 2
incy = 1
alpha = 0.5
do i = 1, 10
  x(i) = 1.0
  y(i) = 1.0
end do
do i = 1, m
  do j = 1, n
    a(i,j) = j
  end do
end do
call sger (m, n, alpha, x, incx, y, incy, a, lda)
print*, `Matrix A: `
do i = 1, m
  print*, (a(i,j), j = 1, n)
end do
end

```

As a result of this program execution, matrix *a* is printed as follows:

Matrix A:

```
1.50000 2.50000 3.50000
1.50000 2.50000 3.50000
```

Example. Using BLAS Level 3 Routine

The following example illustrates a call to the BLAS Level 3 routine `ssymm`. This routine performs a matrix-matrix operation

```
c := alpha*a*b' + beta*c.
```

Parameters

<i>alpha</i>	Specifies a scalar <i>alpha</i> .
<i>beta</i>	Specifies a scalar <i>beta</i> .
<i>a</i>	Symmetric matrix
<i>b</i>	<i>m</i> -by- <i>n</i> matrix
<i>c</i>	<i>m</i> -by- <i>n</i> matrix

```
program symm_main
real a(3,3), b(3,2), c(3,3), alpha, beta
integer m, n, lda, ldb, ldc, i, j
character uplo, side
uplo = 'u'
side = 'l'
m = 3
n = 2
lda = 3
ldb = 3
ldc = 3
alpha = 0.5
beta = 2.0
do i = 1, m
  do j = 1, m
    a(i,j) = 1.0
  end do
end do
do i = 1, m
  do j = 1, n
    c(i,j) = 1.0
    b(i,j) = 2.0
  end do
end do
call ssymm (side, uplo, m, n, alpha,
a, lda, b, ldb, beta, c, ldc)
print*, `Matrix C: `
do i = 1, m
  print*, (c(i,j), j = 1, n)
end do
end
```

As a result of this program execution, matrix *c* is printed as follows:

Matrix C:

```
5.00000 5.00000
```

5.00000 5.00000

5.00000 5.00000

The following example illustrates a call from a C program to the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

Example. Calling a Complex BLAS Level 1 Function from C

In this example, the complex dot product is returned in the structure `c`.

```
#include <cstdio>
#include "mkl blas.h"
#define N 5
void main()
{
    int n, inca = 1, incb = 1, i;
    MKL_Complex16 a[N], b[N], c;
    void zdotc();
    n = N;
    for( i = 0; i < n; i++ ){
        a[i].real = (double)i; a[i].imag = (double)i * 2.0;
        b[i].real = (double)(n - i); b[i].imag = (double)i * 2.0;
    }
    zdotc( &c, &n, a, &inca, b, &incb );
    printf( "The complex dot product is: ( %6.2f, %6.2f )\n", c.real, c.imag );
}
```

NOTE

Instead of calling BLAS directly from C programs, you might wish to use the C interface to the Basic Linear Algebra Subprograms (CBLAS) implemented in Intel® MKL. See [C Interface Conventions](#) for more information.

Fourier Transform Functions Code Examples

This section presents code examples for functions described in the “[FFT Functions](#)” and “[Cluster FFT Functions](#)” sections in the “Fourier Transform Functions” chapter. The examples are grouped in subsections

- [Examples for FFT Functions](#), including [Examples of Using Multi-Threading for FFT Computation](#)
- [Examples for Cluster FFT Functions](#)
- [Auxiliary data transformations](#).

FFT Code Examples

This section presents examples of using the FFT interface functions described in “[Fourier Transform Functions](#)” chapter.

Here are the examples of two one-dimensional computations. These examples use the default settings for all of the configuration parameters, which are specified in “[Configuration Settings](#)”.

In the Fortran examples, the `use mkl_dfti` statement assumes that:

- The `mkl_dfti.f90` module definition file is already compiled.
- The `mkl_dfti.mod` module file is available.

One-dimensional In-place FFT (Fortran Interface)

```
! Fortran example.
! 1D complex to complex, and real to conjugate-even
Use MKL_DFTI
```

```

Complex :: X(32)
Real :: Y(34)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status
!...put input data into X(1),...,X(32); Y(1),...,Y(32)

! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE, &
    DFTI_COMPLEX, 1, 32 )
Status = DftiCommitDescriptor( My_Desc1_Handle )
Status = DftiComputeForward( My_Desc1_Handle, X )
Status = DftiFreeDescriptor( My_Desc1_Handle )
! result is given by {X(1),X(2),...,X(32)}

! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor( My_Desc2_Handle, DFTI_SINGLE, &
    DFTI_REAL, 1, 32 )
Status = DftiCommitDescriptor( My_Desc2_Handle )
Status = DftiComputeForward( My_Desc2_Handle, Y )
Status = DftiFreeDescriptor( My_Desc2_Handle )
! result is given in CCS format.

```

One-dimensional Out-of-place FFT (Fortran Interface)

```

! Fortran example.
! 1D complex to complex, and real to conjugate-even
Use MKL_DFTI
Complex :: X_in(32)
Complex :: X_out(32)
Real :: Y_in(32)
Real :: Y_out(34)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status
...put input data into X_in(1),...,X_in(32); Y_in(1),...,Y_in(32)
! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE,
    DFTI_COMPLEX, 1, 32 )
Status = DftiSetValue( My_Desc1_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE )
Status = DftiCommitDescriptor( My_Desc1_Handle )
Status = DftiComputeForward( My_Desc1_Handle, X_in, X_out )
Status = DftiFreeDescriptor( My_Desc1_Handle )
! result is given by {X_out(1),X_out(2),...,X_out(32)}
! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor( My_Desc2_Handle, DFTI_SINGLE,
    DFTI_REAL, 1, 32 )
Status = DftiSetValue( My_Desc2_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE )
Status = DftiCommitDescriptor( My_Desc2_Handle )
Status = DftiComputeForward( My_Desc2_Handle, Y_in, Y_out )
Status = DftiFreeDescriptor( My_Desc2_Handle )
! result is given by Y_out in CCS format.

```

One-dimensional In-place FFT (C Interface)

```

/* C example, float _Complex is defined in C9X */
#include "mkl_dfti.h"
float _Complex x[32];

```

```

float y[34];
DFTI_DESCRIPTOR_HANDLE my_desc1_handle;
DFTI_DESCRIPTOR_HANDLE my_desc2_handle;
MKL_LONG status;
//...put input data into x[0],...,x[31]; y[0],...,y[31]
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,
                               DFTI_COMPLEX, 1, 32);
status = DftiCommitDescriptor( my_desc1_handle );
status = DftiComputeForward( my_desc1_handle, x );
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is x[0], ..., x[31]*/
status = DftiCreateDescriptor( &my_desc2_handle, DFTI_SINGLE,
                               DFTI_REAL, 1, 32);
status = DftiCommitDescriptor( my_desc2_handle );
status = DftiComputeForward( my_desc2_handle, y );
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is given in CCS format*/

```

One-dimensional Out-of-place FFT (C Interface)

```

/* C example, float _Complex is defined in C9X */
#include "mkl_dfti.h"
float _Complex x_in[32];
float _Complex x_out[32];
float y_in[32];
float y_out[34];

DFTI_DESCRIPTOR_HANDLE my_desc1_handle;
DFTI_DESCRIPTOR_HANDLE my_desc2_handle;
MKL_LONG status;
//...put input data into x_in[0],...,x_in[31]; y_in[0],...,y_in[31]
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,
                               DFTI_COMPLEX, 1, 32);
status = DftiSetValue( my_desc1_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor( my_desc1_handle );
status = DftiComputeForward( my_desc1_handle, x_in, x_out);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is x_out[0], ..., x_out[31]*/
status = DftiCreateDescriptor( &my_desc2_handle, DFTI_SINGLE,
                               DFTI_REAL, 1, 32);
Status = DftiSetValue( My_Desc2_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor( my_desc2_handle );
status = DftiComputeForward( my_desc2_handle, y_in, y_out);
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is given by y_out in CCS format*/

```

Two-dimensional FFT (Fortran Interface)

The following is an example of two simple two-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

```

! Fortran example.
! 2D complex to complex, and real to conjugate-even
Use MKL_DFTI

```

```

Complex :: X_2D(32,100)
Real :: Y_2D(34, 102)
Complex :: X(3200)
Real :: Y(3468)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc1_Handle, My_Desc2_Handle
Integer :: Status, L(2)
!...put input data into X_2D(j,k), Y_2D(j,k), 1<=j<=32,1<=k<=100
!...set L(1) = 32, L(2) = 100
!...the transform is a 32-by-100

! Perform a complex to complex transform
Status = DftiCreateDescriptor( My_Desc1_Handle, DFTI_SINGLE, &
    DFTI_COMPLEX, 2, L)
Status = DftiCommitDescriptor( My_Desc1_Handle)
Status = DftiComputeForward( My_Desc1_Handle, X)
Status = DftiFreeDescriptor( My_Desc1_Handle)
! result is given by X_2D(j,k), 1<=j<=32, 1<=k<=100

! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor( My_Desc2_Handle, DFTI_SINGLE, &
    DFTI_REAL, 2, L)
Status = DftiCommitDescriptor( My_Desc2_Handle)
Status = DftiComputeForward( My_Desc2_Handle, Y)
Status = DftiFreeDescriptor( My_Desc2_Handle)
! result is given by the complex value z(j,k) 1<=j<=32; 1<=k<=100
! and is stored in CCS format

```

Two-dimensional FFT (C Interface)

```

/* C99 example */
#include "mkl_dfti.h"
float _Complex x[32][100];
float y[34][102];
DFTI_DESCRIPTOR_HANDLE my_desc1_handle;
DFTI_DESCRIPTOR_HANDLE my_desc2_handle;
MKL_LONG status, l[2];
//...put input data into x[j][k] 0<=j<=31, 0<=k<=99
//...put input data into y[j][k] 0<=j<=31, 0<=k<=99
l[0] = 32; l[1] = 100;
status = DftiCreateDescriptor( &my_desc1_handle, DFTI_SINGLE,
    DFTI_COMPLEX, 2, l);
status = DftiCommitDescriptor( my_desc1_handle);
status = DftiComputeForward( my_desc1_handle, x);
status = DftiFreeDescriptor(&my_desc1_handle);
/* result is the complex value x[j][k], 0<=j<=31, 0<=k<=99 */
status = DftiCreateDescriptor( &my_desc2_handle, DFTI_SINGLE,
    DFTI_REAL, 2, l);
status = DftiCommitDescriptor( my_desc2_handle);
status = DftiComputeForward( my_desc2_handle, y);
status = DftiFreeDescriptor(&my_desc2_handle);
/* result is the complex value z(j,k) 0<=j<=31; 0<=k<=99
/* and is stored in CCS format*/

```

The following example in C and Fortran demonstrates how you can change the default configuration settings by using the `DftiSetValue` function.

For instance, to preserve the input data after the FFT computation, the configuration of `DFTI_PLACEMENT` should be changed to "not in place" from the default choice of "in place."

The code below illustrates how this can be done:

Changing Default Settings (Fortran Interface)

```
! Fortran example
! 1D complex to complex, not in place
Use MKL_DFTI
Complex :: X_in(32), X_out(32)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status
!...put input data into X_in(j), 1<=j<=32
Status = DftiCreateDescriptor( My_Desc_Handle, & DFTI_SINGLE, DFTI_COMPLEX, 1, 32)
Status = DftiSetValue( My_Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE)
Status = DftiCommitDescriptor( My_Desc_Handle)
Status = DftiComputeForward( My_Desc_Handle, X_in, X_out)
Status = DftiFreeDescriptor (My_Desc_Handle)
! result is X_out(1),X_out(2),...,X_out(32)
```

Changing Default Settings (C Interface)

```
/* C99 example */
#include "mkl_dfti.h"
float _Complex x_in[32], x_out[32];
DFTI_DESCRIPTOR_HANDLE my_desc_handle;
MKL_LONG status;
//...put input data into x_in[j], 0 <= j < 32
status = DftiCreateDescriptor( &my_desc_handle, DFTI_SINGLE,
    DFTI_COMPLEX, 1, 32);
status = DftiSetValue( my_desc_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor( my_desc_handle);
status = DftiComputeForward( my_desc_handle, x_in, x_out);
status = DftiFreeDescriptor(&my_desc_handle);
/* result is x_out[0], x_out[1], ..., x_out[31] */
```

Using Status Checking Functions

This example illustrates the use of status checking functions described in [Chapter 11](#).

```
! Fortran
type(DFTI_DESCRIPTOR), POINTER :: desc
integer status
! ...descriptor creation and other code
status = DftiCommitDescriptor(desc)
if (status .ne. 0) then
  if (.not. DftiErrorClass(status,DFTI_NO_ERROR) then
    print *, 'Error: ', DftiErrorMessage(status)
  endif
```

```
endif

/* C */
DFTI_DESCRIPTOR_HANDLE desc;
MKL_LONG status;
// . . . descriptor creation and other code
status = DftiCommitDescriptor(desc);
if (status && !DftiErrorClass(status,DFTI_NO_ERROR))
{
    printf ('Error: %s\n', DftiErrorMessage(status));
}
```

Computing 2D FFT by One-Dimensional Transforms

Below is an example where a 20-by-40 two-dimensional FFT is computed explicitly using one-dimensional transforms. Notice that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

```
! Fortran
use mkl_dfti
Complex :: X_2D(20,40)
Complex :: X(800)
Equivalence (X_2D, X)
INTEGER :: STRIDE(2)
type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Dim1
type(DFTI_DESCRIPTOR), POINTER :: Desc_Handle_Dim2
! ...
Status = DftiCreateDescriptor(Desc_Handle_Dim1, DFTI_SINGLE,&
                             DFTI_COMPLEX, 1, 20 )
Status = DftiCreateDescriptor(Desc_Handle_Dim2, DFTI_SINGLE,&
                             DFTI_COMPLEX, 1, 40 )
! perform 40 one-dimensional transforms along 1st dimension
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_NUMBER_OF_TRANSFORMS, 40 )
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_INPUT_DISTANCE, 20 )
Status = DftiSetValue( Desc_Handle_Dim1, DFTI_OUTPUT_DISTANCE, 20 )
Status = DftiCommitDescriptor( Desc_Handle_Dim1 )
Status = DftiComputeForward( Desc_Handle_Dim1, X )
! perform 20 one-dimensional transforms along 2nd dimension
Stride(1) = 0; Stride(2) = 20
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_NUMBER_OF_TRANSFORMS, 20 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_DISTANCE, 1 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_DISTANCE, 1 )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_INPUT_STRIDES, Stride )
Status = DftiSetValue( Desc_Handle_Dim2, DFTI_OUTPUT_STRIDES, Stride )
Status = DftiCommitDescriptor( Desc_Handle_Dim2 )
Status = DftiComputeForward( Desc_Handle_Dim2, X )
Status = DftiFreeDescriptor( Desc_Handle_Dim1 )
Status = DftiFreeDescriptor( Desc_Handle_Dim2 )

/* C */
#include "mkl_dfti.h"
float _Complex x[20][40];
MKL_LONG stride[2];
MKL_LONG status;
DFTI_DESCRIPTOR_HANDLE desc_handle_dim1;
```

```

DFTI_DESCRIPTOR_HANDLE desc_handle_dim2;
//...
status = DftiCreateDescriptor( &desc_handle_dim1, DFTI_SINGLE,
    DFTI_COMPLEX, 1, 20 );
status = DftiCreateDescriptor( &desc_handle_dim2, DFTI_SINGLE,
    DFTI_COMPLEX, 1, 40 );

/* perform 40 one-dimensional transforms along 1st dimension */
/* note that the 1st dimension data are not unit-stride */
stride[0] = 0; stride[1] = 40;
status = DftiSetValue( desc_handle_dim1, DFTI_NUMBER_OF_TRANSFORMS, 40 );
status = DftiSetValue( desc_handle_dim1, DFTI_INPUT_DISTANCE, 1 );
status = DftiSetValue( desc_handle_dim1, DFTI_OUTPUT_DISTANCE, 1 );
status = DftiSetValue( desc_handle_dim1, DFTI_INPUT_STRIDES, stride );
status = DftiSetValue( desc_handle_dim1, DFTI_OUTPUT_STRIDES, stride );
status = DftiCommitDescriptor( desc_handle_dim1 );
status = DftiComputeForward( desc_handle_dim1, x );
/* perform 20 one-dimensional transforms along 2nd dimension */
/* note that the 2nd dimension is unit stride */
status = DftiSetValue( desc_handle_dim2, DFTI_NUMBER_OF_TRANSFORMS, 20 );
status = DftiSetValue( desc_handle_dim2,
    DFTI_INPUT_DISTANCE, 40 );
status = DftiSetValue( desc_handle_dim2,
    DFTI_OUTPUT_DISTANCE, 40 );
status = DftiCommitDescriptor( desc_handle_dim2 );
status = DftiComputeForward( desc_handle_dim2, x );
status = DftiFreeDescriptor( &desc_handle_dim1 );
status = DftiFreeDescriptor( &desc_handle_dim2 );

```

The following code illustrates real multi-dimensional transforms with CCE format storage of conjugate-even complex matrix. [Example “Two-Dimensional REAL In-place FFT \(Fortran Interface\)](#)” is two-dimensional in-place transform and [Example “Two-Dimensional REAL Out-of-place FFT \(Fortran Interface\)](#)” is two-dimensional out-of-place transform in Fortran interface. [Example “Three-Dimensional REAL FFT \(C Interface\)](#)” is three-dimensional out-of-place transform in C interface. Note that the data and result parameters in computation functions are all declared as assumed-size rank-1 array `DIMENSION(0:*)`. Therefore two-dimensional array must be transformed to one-dimensional array by `EQUIVALENCE` statement or other facilities of Fortran.

Two-Dimensional REAL In-place FFT (Fortran Interface)

```

! Fortran example.
! 2D and real to conjugate-even
Use MKL_DFTI
Real :: X_2D(34,100) ! 34  = (32/2 + 1)*2
Real :: X(3400)
Equivalence (X_2D, X)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status, L(2)
Integer :: strides_in(3)
Integer :: strides_out(3)
! ...put input data into X_2D(j,k), 1<=j<=32,1<=k<=100
! ...set L(1) = 32, L(2) = 100
! ...set strides_in(1) = 0, strides_in(2) = 1, strides_in(3) = 34
! ...set strides_out(1) = 0, strides_out(2) = 1, strides_out(3) = 17
! ...the transform is a 32-by-100
! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor( My_Desc_Handle, DFTI_SINGLE,&
DFTI_REAL, 2, L )

```

```

Status = DftiSetValue(My_Desc_Handle, DFTI_CONJUGATE_EVEN_STORAGE,&
DFTI_COMPLEX_COMPLEX)
Status = DftiSetValue(My_Desc_Handle, DFTI_INPUT_STRIDES, strides_in)
Status = DftiSetValue(My_Desc_Handle, DFTI_OUTPUT_STRIDES, strides_out)
Status = DftiCommitDescriptor( My_Desc_Handle)
Status = DftiComputeForward( My_Desc_Handle, X )
Status = DftiFreeDescriptor(My_Desc_Handle)
! result is given by the complex value z(j,k) 1<=j<=17; 1<=k<=100 and
! is stored in real matrix X_2D in CCE format.

```

Two-Dimensional REAL Out-of-place FFT (Fortran Interface)

```

! Fortran example.
! 2D and real to conjugate-even
Use MKL_DFTI
Real :: X_2D(32,100)
Complex :: Y_2D(17, 100) ! 17 = 32/2 + 1
Real :: X(3200)
Complex :: Y(1700)
Equivalence (X_2D, X)
Equivalence (Y_2D, Y)
type(DFTI_DESCRIPTOR), POINTER :: My_Desc_Handle
Integer :: Status, L(2)
Integer :: strides_out(3)

! ...put input data into X_2D(j,k), 1<=j=32,1<=k<=100
! ...set L(1) = 32, L(2) = 100
! ...set strides_out(1) = 0, strides_out(2) = 1, strides_out(3) = 17

! ...the transform is a 32-by-100
! Perform a real to complex conjugate-even transform
Status = DftiCreateDescriptor( My_Desc_Handle, DFTI_SINGLE,&
DFTI_REAL, 2, L )
Status = DftiSetValue(My_Desc_Handle,&
DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX)
Status = DftiSetValue( My_Desc_Handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE )
Status = DftiSetValue(My_Desc_Handle,&
DFTI_OUTPUT_STRIDES, strides_out)

Status = DftiCommitDescriptor(My_Desc_Handle)
Status = DftiComputeForward(My_Desc_Handle, X, Y)
Status = DftiFreeDescriptor(My_Desc_Handle)
! result is given by the complex value z(j,k) 1<=j<=17; 1<=k<=100 and
! is stored in complex matrix Y_2D in CCE format.

```

Three-Dimensional REAL FFT (C Interface)

```

/* C99 example */
#include "mkl_dfti.h"
float x[32][100][19];
float _Complex y[32][100][10]; /* 10 = 19/2 + 1 */
DFTI_DESCRIPTOR_HANDLE my_desc_handle;
MKL_LONG status, l[3];
MKL_LONG strides_out[4];

//...put input data into x[j][k][s] 0<=j<=31, 0<=k<=99, 0<=s<=18
l[0] = 32; l[1] = 100; l[2] = 19;

```

```

strides_out[0] = 0; strides_out[1] = 1000;
strides_out[2] = 10; strides_out[3] = 1;

status = DftiCreateDescriptor( &my_desc_handle, DFTI_SINGLE,
DFTI_REAL, 3, 1 );
status = DftiSetValue(my_desc_handle,
DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX);
status = DftiSetValue( my_desc_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE );
status = DftiSetValue(my_desc_handle,
DFTI_OUTPUT_STRIDES, strides_out);

status = DftiCommitDescriptor(my_desc_handle);
status = DftiComputeForward(my_desc_handle, x, y);
status = DftiFreeDescriptor(&my_desc_handle);
/* result is the complex value z(j,k,s) 0<=j<=31; 0<=k<=99, 0<=s<=9
and is stored in complex matrix y in CCE format. */

```

Examples of Using Multi-Threading for FFT Computation

The following sample program shows how to employ internal threading in Intel MKL for FFT computation.

To specify the number of threads inside Intel MKL, use the following settings:

```

set MKL_NUM_THREADS = 1 for one-threaded mode;
set MKL_NUM_THREADS = 4 for multi-threaded mode.

```

Using Intel MKL Internal Threading Mode (C Example)

```

#include "mkl_dfti.h"

int main ()
{
    float x[200][100];
    DFTI_DESCRIPTOR_HANDLE fft;
    MKL_LONG len[2] = {200, 100};
    // initialize x
    DftiCreateDescriptor ( &fft, DFTI_SINGLE, DFTI_REAL, 2, len );
    DftiCommitDescriptor ( fft );
    DftiComputeForward ( fft, x );
    DftiFreeDescriptor ( &fft );
    return 0;
}

```

The following [Example “Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region”](#) and [Example “Using Parallel Mode with Multiple Descriptors Initialized in One Thread”](#) illustrate a parallel customer program with each descriptor instance used only in a single thread.

Specify the number of threads for [Example “Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region”](#) like this:

```

set MKL_NUM_THREADS = 1 for Intel MKL to work in the single-threaded mode (recommended);
set OMP_NUM_THREADS = 4 for the customer program to work in the multi-threaded mode.

```

Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region

Note that in this example, the program can be transformed to become single-threaded at the customer level but using parallel mode within Intel MKL. To achieve this, you need to set the parameter

DFTI_NUMBER_OF_TRANSFORMS = 4 and to set the corresponding parameter DFTI_INPUT_DISTANCE = 5000.

C code for the example is as follows:

```
#include "mkl_dfti.h"
#include <omp.h>
#define ARRAY_LEN(a) sizeof(a)/sizeof(a[0])
int main ()
{
    // 4 OMP threads, each does 2D FFT 50x100 points
    MKL_Complex8 x[4][50][100];
    int nth = ARRAY_LEN(x);
    MKL_LONG len[2] = {ARRAY_LEN(x[0]), ARRAY_LEN(x[0][0])};
    int th;
    // assume x is initialized and do 2D FFTs
#pragma omp parallel for shared(len, x)
    for (th = 0; th < nth; th++)
    {
        DFTI_DESCRIPTOR_HANDLE myFFT;

        DftiCreateDescriptor (&myFFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len);
        DftiCommitDescriptor (myFFT);
        DftiComputeForward (myFFT, x[th]);
        DftiFreeDescriptor (&myFFT);
    }
    return 0;
}
```

Fortran code for the example is as follows:

```
program fft2d_private_descr_main
    use mkl_dfti

    integer nth, len(2)
! 4 OMP threads, each does 2D FFT 50x100 points
    parameter (nth = 4, len = (/50, 100/))
    complex x(len(2)*len(1), nth)

    type(dfti_descriptor), pointer :: myFFT
    integer th, myStatus

! assume x is initialized and do 2D FFTs
 !$OMP PARALLEL DO SHARED(len, x) PRIVATE(myFFT, myStatus)
    do th = 1, nth
        myStatus = DftiCreateDescriptor (myFFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len)
        myStatus = DftiCommitDescriptor (myFFT)
        myStatus = DftiComputeForward (myFFT, x(:, th))
        myStatus = DftiFreeDescriptor (myFFT)
    end do
 !$OMP END PARALLEL DO
end
```

Specify the number of threads for Example “Using Parallel Mode with Multiple Descriptors Initialized in One Thread” like this:

set MKL_NUM_THREADS = 1 for Intel MKL to work in the single-threaded mode (obligatory);
set OMP_NUM_THREADS = 4 for the customer program to work in the multi-threaded mode.

Using Parallel Mode with Multiple Descriptors Initialized in One Thread

C code for the example is as follows:

```
#include "mkl_dfti.h"
#include <omp.h>
#define ARRAY_LEN(a) sizeof(a)/sizeof(a[0])
int main ()
{
    // 4 OMP threads, each does 2D FFT 50x100 points
    MKL_Complex8 x[4][50][100];
    int nth = ARRAY_LEN(x);
    MKL_LONG len[2] = {ARRAY_LEN(x[0]), ARRAY_LEN(x[0][0])};
    DFTI_DESCRIPTOR_HANDLE FFT[ARRAY_LEN(x)];
    int th;

    for (th = 0; th < nth; th++)
        DftiCreateDescriptor (&FFT[th], DFTI_SINGLE, DFTI_COMPLEX, 2, len);
    for (th = 0; th < nth; th++)
        DftiCommitDescriptor (FFT[th]);
    // assume x is initialized and do 2D FFTs
#pragma omp parallel for shared(FFT, x)
    for (th = 0; th < nth; th++)
        DftiComputeForward (FFT[th], x[th]);
    for (th = 0; th < nth; th++)
        DftiFreeDescriptor (&FFT[th]);
    return 0;
}
```

Fortran code for the example is as follows:

```
program fft2d_array_descr_main
    use mkl_dfti

    integer nth, len(2)
! 4 OMP threads, each does 2D FFT 50x100 points
    parameter (nth = 4, len = (/50, 100/))
    complex x(len(2)*len(1), nth)

    type thread_data
        type(dfti_descriptor), pointer :: FFT
    end type thread_data
    type(thread_data) :: workload(nth)

    integer th, status, myStatus

    do th = 1, nth
        status = DftiCreateDescriptor (workload(th)%FFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len)
        status = DftiCommitDescriptor (workload(th)%FFT)
    end do
    ! assume x is initialized and do 2D FFTs
    !$OMP PARALLEL DO SHARED(len, x, workload) PRIVATE(myStatus)
        do th = 1, nth
            myStatus = DftiComputeForward (workload(th)%FFT, x(:, th))
        end do
    !$OMP END PARALLEL DO
    do th = 1, nth
        status = DftiFreeDescriptor (workload(th)%FFT)
    end do
end
```

The following Example “Using Parallel Mode with a Common Descriptor” illustrates a parallel customer program with a common descriptor used in several threads.

Using Parallel Mode with a Common Descriptor

C code for the example is as follows:

```
#include "mkl_dfti.h"
#include <omp.h>
#define ARRAY_LEN(a) sizeof(a)/sizeof(a[0])
int main ()
{
    // 4 OMP threads, each does 2D FFT 50x100 points
    MKL_Complex8 x[4][50][100];
    int nth = ARRAY_LEN(x);
    MKL_LONG len[2] = {ARRAY_LEN(x[0]), ARRAY_LEN(x[0][0])};
    DFTI_DESCRIPTOR_HANDLE FFT;
    int th;

    DftiCreateDescriptor (&FFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len);
    DftiCommitDescriptor (FFT);
    // assume x is initialized and do 2D FFTs
#pragma omp parallel for shared(FFT, x)
    for (th = 0; th < nth; th++)
        DftiComputeForward (FFT, x[th]);
    DftiFreeDescriptor (&FFT);
    return 0;
}
```

Fortran code for the example is as follows:

```
program fft2d_shared_descr_main
  use mkl_dfti

  integer nth, len(2)
! 4 OMP threads, each does 2D FFT 50x100 points
  parameter (nth = 4, len = (/50, 100/))
  complex x(len(2)*len(1), nth)
  type(dfti_descriptor), pointer :: FFT

  integer th, status, myStatus

  status = DftiCreateDescriptor (FFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len)
  status = DftiCommitDescriptor (FFT)
! assume x is initialized and do 2D FFTs
 !$OMP PARALLEL DO SHARED(len, x, FFT) PRIVATE(myStatus)
  do th = 1, nth
    myStatus = DftiComputeForward (FFT, x(:, th))
  end do
 !$OMP END PARALLEL DO
  status = DftiFreeDescriptor (FFT)
end
```

Examples for Cluster FFT Functions

The following C example computes a 2-dimensional out-of-place FFT using the cluster FFT interface:

2D Out-of-place Cluster FFT Computation

```
DFTI_DESCRIPTOR_DM_HANDLE desc;
MKL_LONG len[2], v, i, j, n, s;
Complex *in, *out;
```

```

MPI_Init(...);

// Create descriptor for 2D FFT
len[0]=nx;
len[1]=ny;
DftiCreateDescriptorDM(MPI_COMM_WORLD,&desc,DFTI_DOUBLE,DFTI_COMPLEX,2,len);
// Ask necessary length of in and out arrays and allocate memory
DftiGetValueDM(desc,CDFL_LOCAL_SIZE,&v);
in=(Complex*)malloc(v*sizeof(Complex));
out=(Complex*)malloc(v*sizeof(Complex));
// Fill local array with initial data. Current process performs n rows,
// 0 row of in corresponds to s row of virtual global array
DftiGetValueDM(desc,CDFL_LOCAL_NX,&n);
DftiGetValueDM(desc,CDFL_LOCAL_X_START,&s);
// Virtual global array globalIN is defined by function f as
// globalIN[i*ny+j]=f(i,j)
for(i=0;i<n;i++)
    for(j=0;j<ny;j++) in[i*ny+j]=f(i+s,j);
// Set that we want out-of-place transform (default is DFTI_INPLACE)
DftiSetValueDM(desc,DFTI_PLACEMENT,DFTI_NOT_INPLACE);
// Commit descriptor, calculate FFT, free descriptor
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc,in,out);
// Virtual global array globalOUT is defined by function g as
// globalOUT[i*ny+j]=g(i,j)
// Now out contains result of FFT. out[i*ny+j]=g(i+s,j)
DftiFreeDescriptorDM(&desc);
free(in);
free(out);
MPI_Finalize();

```

1D In-place Cluster FFT Computations

The C example below illustrates one-dimensional in-place cluster FFT computations effected with a user-defined workspace:

```

DFTI_DESCRIPTOR_DM_HANDLE desc;
MKL_LONG len,v,i,n_out,s_out;
Complex *in,*work;

MPI_Init(...);
// Create descriptor for 1D FFT
DftiCreateDescriptorDM(MPI_COMM_WORLD,&desc,DFTI_DOUBLE,DFTI_COMPLEX,1,len);
// Ask necessary length of array and workspace and allocate memory
DftiGetValueDM(desc,CDFL_LOCAL_SIZE,&v);
in=(Complex*)malloc(v*sizeof(Complex));
work=(Complex*)malloc(v*sizeof(Complex));
// Fill local array with initial data. Local array has n elements,
// 0 element of in corresponds to s element of virtual global array
DftiGetValueDM(desc,CDFL_LOCAL_NX,&n);
DftiGetValueDM(desc,CDFL_LOCAL_X_START,&s);
// Set work array as a workspace
DftiSetValueDM(desc,CDFL_WORKSPACE,work);
// Virtual global array globalIN is defined by function f as globalIN[i]=f(i)
for(i=0;i<n;i++) in[i]=f(i+s);
// Commit descriptor, calculate FFT, free descriptor
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc,in);
DftiGetValueDM(desc,CDFL_LOCAL_OUT_NX,&n_out);

```

```

DftiGetValueDM(desc,CDFT_LOCAL_OUT_X_START,&s_out);
// Virtual global array globalOUT is defined by function g as globalOUT[i]=g(i)
// Now in contains result of FFT. Local array has n_out elements,
// 0 element of in corresponds to s_out element of virtual global array.
// in[i]==g(i+s_out)
DftiFreeDescriptorDM(&desc);
free(in);
free(work);
MPI_Finalize();

```

Auxiliary Data Transformations

This section presents C examples for conversion from the Cartesian to polar representation of complex data and vice versa.

Conversion from Cartesian to polar representation of complex data

```

// Cartesian->polar conversion of complex data
// Cartesian representation: z = re + I*im
// Polar representation: z = r * exp( I*phi )
#include <mkl_vml.h>

void
variant1_Cartesian2Polar(int n,const double *re,const double *im,
                         double *r,double *phi)
{
    vdHypot(n,re,im,r);           // compute radii r[]
    vdAtan2(n,im,re,phi);        // compute phases phi[]
}

void
variant2_Cartesian2Polar(int n,const MKL_Complex16 *z,double *r,double *phi,
                         double *temp_re,double *temp_im)
{
    vzAbs(n,z,r);               // compute radii r[]
    vdPackI(n, (double*)z + 0, 2, temp_re);
    vdPackI(n, (double*)z + 1, 2, temp_im);
    vdAtan2(n,temp_im,temp_re,phi); // compute phases phi[]
}

```

Conversion from polar to Cartesian representation of complex data

```

// Polar->Cartesian conversion of complex data.
// Polar representation: z = r * exp( I*phi )
// Cartesian representation: z = re + I*im
#include <mkl_vml.h>

void
variant1_Polar2Cartesian(int n,const double *r,const double *phi,
                         double *re,double *im)
{
    vdSinCos(n,phi,im,re);       // compute direction, i.e. z[]/abs(z[])
    vdMul(n,r,re,re);           // scale real part
    vdMul(n,r,im,im);           // scale imaginary part
}

void

```

```
variant2_Polar2Cartesian(int n,const double *r,const double *phi,
                         MKL_Complex16 *z,
                         double *temp_re,double *temp_im)
{
    vdSinCos(n,phi,temp_im,temp_re); // compute direction, i.e. z[]/abs(z[])
    vdMul(n,r,temp_im,temp_im); // scale imaginary part
    vdMul(n,r,temp_re,temp_re); // scale real part
    vdUnpackI(n,temp_re,(double*)z + 0, 2); // fill in result.re
    vdUnpackI(n,temp_im,(double*)z + 1, 2); // fill in result.im
}
```

Bibliography

For more information about the BLAS, Sparse BLAS, LAPACK, ScaLAPACK, Sparse Solver, Extended Eigensolver, VML, VSL, FFT, and Non-Linear Optimization Solvers functionality, refer to the following publications:

- **BLAS Level 1**

C. Lawson, R. Hanson, D. Kincaid, and F. Krough. *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software, Vol.5, No.3 (September 1979) 308-325.

- **BLAS Level 2**

J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.14, No.1 (March 1988) 1-32.

- **BLAS Level 3**

J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software (December 1989).

- **Sparse BLAS**

D. Dodson, R. Grimes, and J. Lewis. *Sparse Extensions to the FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Math Software, Vol.17, No.2 (June 1991).

D. Dodson, R. Grimes, and J. Lewis. *Algorithm 692: Model Implementation and Test Package for the Sparse Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol.17, No.2 (June 1991).

[Duff86] I.S.Duff, A.M.Erisman, and J.K.Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, UK, 1986.

[CXML01] *Compaq Extended Math Library*. Reference Guide, Oct.2001.

[Rem05] K.Remington. *A NIST FORTRAN Sparse Blas User's Guide*. (available on <http://math.nist.gov/~KRemington/fspblas/>)

[Saad94] Y.Saad. *SPARSKIT: A Basic Tool-kit for Sparse Matrix Computation*. Version 2, 1994. (<http://www.cs.umn.edu/~saad>)

[Saad96] Y.Saad. *Iterative Methods for Linear Systems*. PWS Publishing, Boston, 1996.

- **LAPACK**

[AndaPark94] A. A. Anda and H. Park. *Fast plane rotations with dynamic scaling*, SIAM J. matrix Anal. Appl., Vol. 15 (1994), pp. 162-174.

[Baudin12] M. Baudin, R. Smith. *A Robust Complex Division in Scilab*, available from <http://www.arxiv.org>, arXiv:1210.4539v2 (2012).

[Bischof00] C. H. Bischof, B. Lang, and X. Sun. *Algorithm 807: The SBR toolbox-software for successive band reduction*, ACM Transactions on Mathematical Software, Vol. 26, No. 4, pages 602-616, December 2000.

[Demmel92] J. Demmel and K. Veselic. *Jacobi's method is more accurate than QR*, SIAM J. Matrix Anal. Appl. 13(1992):1204-1246.

[deRijk98] P. P. M. De Rijk. *A one-sided Jacobi algorithm for computing the singular value decomposition on a vector computer*, SIAM J. Sci. Stat. Comp., Vol. 10 (1998), pp. 359-371.

- [Dhillon04] I. Dhillon, B. Parlett. *Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices*, Linear Algebra and its Applications, 387(1), pp. 1-28, August 2004.
- [Dhillon04-02] I. Dhillon, B. Parlett. *Orthogonal Eigenvectors and * Relative Gaps*, SIAM Journal on Matrix Analysis and Applications, Vol. 25, 2004. (Also LAPACK Working Note 154.)
- [Dhillon97] I. Dhillon. *A new O(n^2) algorithm for the symmetric tridiagonal eigenvalue/eigenvector problem*, Computer Science Division Technical Report No. UCB/CSD-97-971, UC Berkeley, May 1997.
- [Drmac08-1] Z. Drmac and K. Veselic. *New fast and accurate Jacobi SVD algorithm I*, SIAM J. Matrix Anal. Appl. Vol. 35, No. 2 (2008), pp. 1322-1342. LAPACK Working note 169.
- [Drmac08-2] Z. Drmac and K. Veselic. *New fast and accurate Jacobi SVD algorithm II*, SIAM J. Matrix Anal. Appl. Vol. 35, No. 2 (2008), pp. 1343-1362. LAPACK Working note 170.
- [Drmac08-3] Z. Drmac and K. Bujanovic. *On the failure of rank-revealing QR factorization software - a case study*, ACM Trans. Math. Softw. Vol. 35, No 2 (2008), pp. 1-28. LAPACK Working note 176.
- [Drmac08-4] Z. Drmac. *Implementation of Jacobi rotations for accurate singular value computation in floating point arithmetic*, SIAM J. Sci. Comp., Vol. 18 (1997), pp. 1200-1222.
- [Golub96] G. Golub and C. Van Loan. *Matrix Computations*, Johns Hopkins University Press, Baltimore, third edition, 1996.
- [LUG] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*, Third Edition, Society for Industrial and Applied Mathematics (SIAM), 1999.
- [Kahan66] W. Kahan. *Accurate Eigenvalues of a Symmetric Tridiagonal Matrix*, Report CS41, Computer Science Dept., Stanford University, July 21, 1966.
- [Marques06] O. Marques, E.J. Riedy, and Ch. Voemel. *Benefits of IEEE-754 Features in Modern Symmetric Tridiagonal Eigensolvers*, SIAM Journal on Scientific Computing, Vol. 28, No. 5, 2006. (Tech report version in LAPACK Working Note 172 with the same title.)
- [Sutton09] Brian D. Sutton. *Computing the complete CS decomposition*, Numer. Algorithms, 50(1):33-65, 2009.
- **ScaLAPACK**
- [SLUG] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics (SIAM), 1997.
- **Sparse Solver**
- [Duff99] I. S. Duff and J. Koster. *The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices*. SIAM J. Matrix Analysis and Applications, 20(4):889-901, 1999.

- [Dong95] J. Dongarra, V.Eijkhout, A.Kalhan. *Reverse Communication Interface for Linear Algebra Templates for Iterative Methods*. UT-CS-95-291, May 1995. <http://www.netlib.org/lapack/lawnspdf/lawn99.pdf>
- [Karypis98] G. Karypis and V. Kumar. *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*. SIAM Journal on Scientific Computing, 20(1): 359-392, 1998.
- [Li99] X.S. Li and J.W. Demmel. *A Scalable Sparse Direct Solver Using Static Pivoting*. In Proceeding of the 9th SIAM conference on Parallel Processing for Scientific Computing, San Antonio, Texas, March 22-34, 1999.
- [Liu85] J.W.H. Liu. *Modification of the Minimum-Degree algorithm by multiple elimination*. ACM Transactions on Mathematical Software, 11(2):141-153, 1985.
- [Menon98] R. Menon L. Dagnum. *OpenMP: An Industry-Standard API for Shared-Memory Programming*. IEEE Computational Science & Engineering, 1:46-55, 1998. <http://www.openmp.org>.
- [Saad03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd edition, SIAM, Philadelphia, PA, 2003.
- [Schenk00] O. Schenk. *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors*. PhD thesis, ETH Zurich, 2000.
- [Schenk00-2] O. Schenk, K. Gartner, and W. Fichtner. *Efficient Sparse LU Factorization with Left-right Looking Strategy on Shared Memory Multiprocessors*. BIT, 40(1): 158-176, 2000.
- [Schenk01] O. Schenk and K. Gartner. *Sparse Factorization with Two-Level Scheduling in PARDISO*. In Proceeding of the 10th SIAM conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, March 12-14, 2001.
- [Schenk02] O. Schenk and K. Gartner. *Two-level scheduling in PARDISO: Improved Scalability on Shared Memory Multiprocessing Systems*. Parallel Computing, 28:187-197, 2002.
- [Schenk03] O. Schenk and K. Gartner. *Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO*. Journal of Future Generation Computer Systems, 20(3):475-487, 2004.
- [Schenk04] O. Schenk and K. Gartner. *On Fast Factorization Pivoting Methods for Sparse Symmetric Indefinite Systems*. Technical Report, Department of Computer Science, University of Basel, 2004, submitted.
- [Sonneveld89] P. Sonneveld. *CGS, a Fast Lanczos-Type Solver for Nonsymmetric Linear Systems*. SIAM Journal on Scientific and Statistical Computing, 10:36-52, 1989.
- [Young71] D.M. Young. *Iterative Solution of Large Linear Systems*. New York, Academic Press, Inc., 1971.

- **Extended Eigensolver**

- [Polizzi09] E. Polizzi, *Density-Matrix-Based Algorithms for Solving Eigenvalue Problems*, Phys. Rev. B. Vol. 79, 115112, 2009.
- [Polizzi12] E. Polizzi, *A High-Performance Numerical Library for Solving Eigenvalue Problems: FEAST Solver v2.0 User's Guide*, arxiv.org/abs/1203.4031, 2012.

- [Bai00] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe and H. van der Vorst, editors, *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia, 2000.
- [Sleijpen96] G. L. G. Sleijpen and H. A. van der Vorst. *A Jacobi-Davidson iteration method for linear eigenvalue problems*. SIAM J. Matrix Anal. Appl., 17:401-425, 1996.
- **VSL**
- [AVX] Intel. *Intel® Advanced Vector Extensions Programming Reference*. (<http://software.intel.com/file/36945>)
- [Billor00] Nedret Billor, Ali S. Hadib, and Paul F. Velleman. *BACON: blocked adaptive computationally efficient outlier nominators*. Computational Statistics & Data Analysis, 34, 279-298, 2000.
- [Bratley87] Bratley P., Fox B.L., and Schrage L.E. *A Guide to Simulation*. 2nd edition. Springer-Verlag, New York, 1987.
- [Bratley88] Bratley P. and Fox B.L. *Implementing Sobol's Quasirandom Sequence Generator*, ACM Transactions on Mathematical Software, Vol. 14, No. 1, Pages 88-100, March 1988.
- [Bratley92] Bratley P., Fox B.L., and Niederreiter H. *Implementation and Tests of Low-Discrepancy Sequences*, ACM Transactions on Modeling and Computer Simulation, Vol. 2, No. 3, Pages 195-213, July 1992.
- [BMT] Intel. *Bull Mountain Technology Software Implementation Guide*. (<http://software.intel.com/file/37157>)
- [Coddington94] Coddington, P. D. *Analysis of Random Number Generators Using Monte Carlo Simulation*. Int. J. Mod. Phys. C-5, 547, 1994.
- [Gentle98] Gentle, James E. *Random Number Generation and Monte Carlo Methods*, Springer-Verlag New York, Inc., 1998.
- [IntelSWMan] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 3 vols. (<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>)
- [L'Ecuyer94] L'Ecuyer, Pierre. *Uniform Random Number Generation*. Annals of Operations Research, 53, 77-120, 1994.
- [L'Ecuyer99] L'Ecuyer, Pierre. *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure*. Mathematics of Computation, 68, 225, 249-260, 1999.
- [L'Ecuyer99a] L'Ecuyer, Pierre. *Good Parameter Sets for Combined Multiple Recursive Random Number Generators*. Operations Research, 47, 1, 159-164, 1999.
- [L'Ecuyer01] L'Ecuyer, Pierre. *Software for Uniform Random Number Generation: Distinguishing the Good and the Bad*. Proceedings of the 2001 Winter Simulation Conference, IEEE Press, 95-105, Dec. 2001.
- [Kirkpatrick81] Kirkpatrick, S., and Stoll, E. *A Very Fast Shift-Register Sequence Random Number Generator*. Journal of Computational Physics, V. 40. 517-526, 1981.
- [Knuth81] Knuth, Donald E. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. 2nd edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.
- [Maronna02] Maronna, R.A., and Zamar, R.H., *Robust Multivariate Estimates for High-Dimensional Datasets*, Technometrics, 44, 307-317, 2002.

- [Matsumoto98] Matsumoto, M., and Nishimura, T. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*, ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, Pages 3-30, January 1998.
- [Matsumoto00] Matsumoto, M., and Nishimura, T. *Dynamic Creation of Pseudorandom Number Generators*, 56-69, in: Monte Carlo and Quasi-Monte Carlo Methods 1998, Ed. Niederreiter, H. and Spanier, J., Springer 2000, <http://www.math.sci.hiroshima-u.ac.jp/%7Em-mat/MT/DC/dc.html>.
- [NAG] NAG Numerical Libraries. http://www.nag.co.uk/numeric_numerical_libraries.asp
- [Rocke96] David M. Rocke, *Robustness properties of S-estimators of multivariate location and shape in high dimension*. The Annals of Statistics, 24(3), 1327-1345, 1996.
- [Saito08] Saito, M., and Matsumoto, M. *SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*. Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, Pages 607 – 622, 2008.
<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/earticles.html>
- [Schafer97] Schafer, J.L., *Analysis of Incomplete Multivariate Data*. Chapman & Hall, 1997.
- [Sobol76] Sobol, I.M., and Levitan, Yu.L. *The production of points uniformly distributed in a multidimensional cube*. Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976 (In Russian).
- [SSL Notes] *Intel® MKL Summary Statistics Library Application Notes*, a document present on the Intel® MKL product at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>
- [VSL Notes] *Intel® MKL Vector Statistical Library Notes*, a document present on the Intel® MKL product at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>
- [VSL Data] *Intel® MKL Vector Statistical Library Performance*, a document present on the Intel® MKL product at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>
- **VML**
- [C99] ISO/IEC 9899:1999/Cor 3:2007. Programming languages -- C.
- [Muller97] J.M.Muller. *Elementary functions: algorithms and implementation*, Birkhauser Boston, 1997.
- [IEEE754] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-2008.
- [VML Data] *Intel® MKL Vector Math Library Performance and Accuracy*, a document present on the Intel® MKL product at <http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/>
- **FFT**
- [1] E. Oran Brigham, *The Fast Fourier Transform and Its Applications*, Prentice Hall, New Jersey, 1988.
- [2] Athanasios Papoulis, *The Fourier Integral and its Applications*, 2nd edition, McGraw-Hill, New York, 1984.

[3] Ping Tak Peter Tang, *DFTI - a new interface for Fast Fourier Transform libraries*, ACM Transactions on Mathematical Software, Vol. 31, Issue 4, Pages 475 - 507, 2005.

[4] Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.

• Optimization Solvers

[Conn00] A. R. Conn, N. I.M. Gould, P. L. Toint. *Trust-region Methods*. SIAM Society for Industrial & Applied Mathematics, Englewood Cliffs, New Jersey, MPS-SIAM Series on Optimization edition, 2000.

[Dong95] J. Dongarra, V. Eijkhout, A. Kalhan. *Reverse communication interface for linear algebra templates for iterative methods*. 1995.

• Data Fitting Functions

[deBoor2001] Carl deBoor. *A Practical Guide to Splines*. Revised Edition. Springer-Verlag New York Berlin Heidelberg, 2001.

[Schumaker2007] Larry L Schumaker. *Spline Functions: Basic Theory*. 3rd Edition. Cambridge University Press, Cambridge, 2007.

[StechSub76] S.B. Stechkin, and Yu Subbotin. *Splines in Numerical Mathematics*. Izd. Nauka, Moscow, 1976.

For a reference implementation of BLAS, sparse BLAS, LAPACK, and ScaLAPACK packages visit
www.netlib.org.



Glossary

A^H	Denotes the conjugate transpose of a general matrix A . See also conjugate matrix.
A^T	Denotes the transpose of a general matrix A . See also transpose.
band matrix	A general m -by- n matrix A such that $a_{ij} = 0$ for $ i - j > l$, where $1 \leq l \leq \min(m, n)$. For example, any tridiagonal matrix is a band matrix.
band storage	A special storage scheme for band matrices. A matrix is stored in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and <i>diagonals</i> of the matrix are stored in rows of the array.
BLAS	Abbreviation for Basic Linear Algebra Subprograms. These subprograms implement vector, matrix-vector, and matrix-matrix operations.
BRNG	Abbreviation for Basic Random Number Generator. Basic random number generators are pseudorandom number generators imitating i.i.d. random number sequences of uniform distribution. Distributions other than uniform are generated by applying different transformation techniques to the sequences of random numbers of uniform distribution.
BRNG registration	Standardized mechanism that allows a user to include a user-designed BRNG into the VSL and use it along with the predefined VSL basic generators.
Bunch-Kaufman factorization	Representation of a real symmetric or complex Hermitian matrix A in the form $A = PUDU^H P^T$ (or $A = PLDL^H P^T$) where P is a permutation matrix, U and L are upper and lower triangular matrices with unit diagonal, and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D .
c	When found as the first letter of routine names, c indicates the usage of single-precision complex data type.
CBLAS	C interface to the BLAS. See BLAS.
CDF	Cumulative Distribution Function. The function that determines probability distribution for univariate or multivariate random variable X . For univariate distribution the cumulative distribution function is the function of real argument x , which for every x takes a value equal to probability of the event $A: X \leq x$. For multivariate distribution the cumulative distribution function is the function of a real vector $x = (x_1, x_2, \dots, x_n)$, which, for every x , takes a value equal to probability of the event $A = (X_1 \leq x_1 \& X_2 \leq x_2, \& \dots, \& X_n \leq x_n)$.

Cholesky factorization	Representation of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix A in the form $A = U^H U$ or $A = LL^H$, where L is a lower triangular matrix and U is an upper triangular matrix.
condition number	The number $\kappa(A)$ defined for a given square matrix A as follows: $\kappa(A) = A \cdot A^{-1} $.
conjugate matrix	The matrix A^H defined for a given general matrix A as follows: $(A^H)_{ij} = (a_{ji})^*$.
conjugate number	The conjugate of a complex number $z = a + bi$ is $z^* = a - bi$.
d	When found as the first letter of routine names, d indicates the usage of double-precision real data type.
dot product	The number denoted $x \cdot y$ and defined for given vectors x and y as follows: $x \cdot y = \sum_i x_i y_i$. Here x_i and y_i stand for the i -th elements of x and y , respectively.
double precision	A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $2.23 \times 10^{-308} < x < 1.79 \times 10^{308}$. For this data type, the machine precision ϵ is approximately 10^{-15} , which means that double-precision numbers usually contain no more than 15 significant decimal digits. For more information, refer to <i>Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture</i> .
eigenvalue	See eigenvalue problem.
eigenvalue problem	A problem of finding non-zero vectors x and numbers λ (for a given square matrix A) such that $Ax = \lambda x$. Here the numbers λ are called the eigenvalues of the matrix A and the vectors x are called the eigenvectors of the matrix A .
eigenvector	See eigenvalue problem.
elementary reflector(Householder matrix)	Matrix of a general form $H = I - \tau v v^T$, where v is a column vector and τ is a scalar. In LAPACK elementary reflectors are used, for example, to represent the matrix Q in the QR factorization (the matrix Q is represented as a product of elementary reflectors).
factorization	Representation of a matrix as a product of matrices. See also Bunch-Kaufman factorization, Cholesky factorization, LU factorization, LQ factorization, QR factorization, Schur factorization.
FFTs	Abbreviation for Fast Fourier Transforms. See Chapter 11 of this book.
full storage	A storage scheme allowing you to store matrices of any kind. A matrix A is stored in a two-dimensional array a , with the matrix element a_{ij} stored in the array element $a(i, j)$.
Hermitian matrix	A square matrix A that is equal to its conjugate matrix A^H . The conjugate A^H is defined as follows: $(A^H)_{ij} = (a_{ji})^*$.
I	See identity matrix.
identity matrix	A square matrix I whose diagonal elements are 1, and off-diagonal elements are 0. For any matrix A , $AI = A$ and $IA = A$.

i.i.d.	Independent Identically Distributed.
in-place	Qualifier of an operation. A function that performs its operation in-place takes its input from an array and returns its output to the same array.
Intel MKL	Abbreviation for Intel® Math Kernel Library.
inverse matrix	The matrix denoted as A^{-1} and defined for a given square matrix A as follows: $AA^{-1} = A^{-1}A = I$. A^{-1} does not exist for singular matrices A .
LQ factorization	Representation of an m -by- n matrix A as $A = LQ$ or $A = (L \ 0)Q$. Here Q is an n -by- n orthogonal (unitary) matrix. For $m \leq n$, L is an m -by- m lower triangular matrix with real diagonal elements; for $m > n$,
	where L_1 is an n -by- n lower triangular matrix, and L_2 is a rectangular matrix.
LU factorization	Representation of a general m -by- n matrix A as $A = PLU$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and U is upper triangular (upper trapezoidal if $m < n$).
machine precision	The number ε determining the precision of the machine representation of real numbers. For Intel® architecture, the machine precision is approximately 10^{-7} for single-precision data, and approximately 10^{-15} for double-precision data. The precision also determines the number of significant decimal digits in the machine representation of real numbers. See also double precision and single precision.
MPI	Message Passing Interface. This standard defines the user interface and functionality for a wide range of message-passing capabilities in parallel computing.
MPICH	A freely available, portable implementation of MPI standard for message-passing libraries.
orthogonal matrix	A real square matrix A whose transpose and inverse are equal, that is, $A^T = A^{-1}$, and therefore $AA^T = A^TA = I$. All eigenvalues of an orthogonal matrix have the absolute value 1.
packed storage	A storage scheme allowing you to store symmetric, Hermitian, or triangular matrices more compactly. The upper or lower triangle of a matrix is packed by columns in a one-dimensional array.
PDF	Probability Density Function. The function that determines probability distribution for univariate or multivariate continuous random variable X . The probability density function $f(x)$ is closely related with the cumulative distribution function $F(x)$. For univariate distribution the relation is

$$F(x) = \int_{-\infty}^x f(t)dt .$$

For multivariate distribution the relation is

$$F(x_1, x_2, \dots, x_n) = \int_{-\infty}^{x_1} \int_{-\infty}^{x_2} \dots \int_{-\infty}^{x_n} f(t_1, t_2, \dots, t_n) dt_1 dt_2 \dots dt_n$$

positive-definite matrix

A square matrix A such that $\mathbf{Ax} \cdot \mathbf{x} > 0$ for any non-zero vector x . Here \cdot denotes the dot product.

pseudorandom number generator

A completely deterministic algorithm that imitates truly random sequences.

QR factorization

Representation of an m -by- n matrix A as $A = QR$, where Q is an m -by- m orthogonal (unitary) matrix, and R is n -by- n upper triangular with real diagonal elements (if $m \geq n$) or trapezoidal (if $m < n$) matrix.

random stream

An abstract source of independent identically distributed random numbers of uniform distribution. In this manual a random stream points to a structure that uniquely defines a random number sequence generated by a basic generator associated with a given random stream.

RNG

Abbreviation for Random Number Generator. In this manual the term "random number generators" stands for pseudorandom number generators, that is, generators based on completely deterministic algorithms imitating truly random sequences.

Rectangular Full Packed (RFP) storage

A storage scheme combining the full and packed storage schemes for the upper or lower triangle of the matrix. This combination enables using half of the full storage as packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels as the full storage.

s

When found as the first letter of routine names, s indicates the usage of single-precision real data type.

ScaLAPACK

Stands for Scalable Linear Algebra PACKage.

Schur factorization

Representation of a square matrix A in the form $A = ZTZ^H$. Here T is an upper quasi-triangular matrix (for complex A , triangular matrix) called the Schur form of A ; the matrix Z is orthogonal (for complex A , unitary). Columns of Z are called Schur vectors.

single precision

A floating-point data type. On Intel® processors, this data type allows you to store real numbers x such that $1.18 \times 10^{-38} < |x| < 3.40 \times 10^{38}$. For this data type, the machine precision (ϵ) is approximately 10^{-7} , which means that single-precision numbers usually contain no more than 7 significant decimal digits. For more information, refer to *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

singular matrix

A matrix whose determinant is zero. If A is a singular matrix, the inverse A^{-1} does not exist, and the system of equations $Ax = b$ does not have a unique solution (that is, there exist no solutions or an infinite number of solutions).

singular value	The numbers defined for a given general matrix A as the eigenvalues of the matrix AA^H . See also SVD.
SMP	Abbreviation for Symmetric MultiProcessing. The MKL offers performance gains through parallelism provided by the SMP feature.
sparse BLAS	Routines performing basic vector operations on sparse vectors. Sparse BLAS routines take advantage of vectors' sparsity: they allow you to store only non-zero elements of vectors. See BLAS.
sparse vectors	Vectors in which most of the components are zeros.
storage scheme	The way of storing matrices. See full storage, packed storage, and band storage.
SVD	Abbreviation for Singular Value Decomposition. See also Singular value decomposition section in Chapter 5.
symmetric matrix	A square matrix A such that $a_{ij} = a_{ji}$.
transpose	The transpose of a given matrix A is a matrix A^T such that $(A^T)_{ij} = a_{ji}$ (rows of A become columns of A^T , and columns of A become rows of A^T).
trapezoidal matrix	A matrix A such that $A = (A_1 A_2)$, where A_1 is an upper triangular matrix, A_2 is a rectangular matrix.
triangular matrix	A matrix A is called an upper (lower) triangular matrix if all its subdiagonal elements (superdiagonal elements) are zeros. Thus, for an upper triangular matrix $a_{ij} = 0$ when $i > j$; for a lower triangular matrix $a_{ij} = 0$ when $i < j$.
tridiagonal matrix	A matrix whose non-zero elements are in three diagonals only: the leading diagonal, the first subdiagonal, and the first super-diagonal.
unitary matrix	A complex square matrix A whose conjugate and inverse are equal, that is, that is, $A^H = A^{-1}$, and therefore $AA^H = A^HA = I$. All eigenvalues of a unitary matrix have the absolute value 1.
VML	Abbreviation for Vector Mathematical Library. See Chapter 9 of this book.
VSL	Abbreviation for Vector Statistical Library. See Chapter 10 of this book.
z	When found as the first letter of routine names, z indicates the usage of double-precision complex data type.

Index

?_backward_trig_transform 3102
?_commit_Helmholtz_2D 3122
?_commit_Helmholtz_3D 3122
?_commit_sph_np 3130
?_commit_sph_p 3130
?_commit_trig_transform 3098
?_forward_trig_transform 3101
?_Helmholtz_2D 3125
?_Helmholtz_3D 3125
?_init_Helmholtz_2D 3119
?_init_Helmholtz_3D 3119
?_init_sph_np 3129
?_init_sph_p 3129
?_init_trig_transform 3097
?_sph_np 3132
?_sph_p 3132
?asum 56
?axpby 388
?axpy 57
?axpyi 169
?bdssdc 902
?bdsqr 898
?cabs1 79
?ConvExec 2851
?ConvExec1D 2855
?ConvExecX 2858
?ConvExecX1D 2862
?ConvNewTask 2833
?ConvNewTask1D 2835
?ConvNewTaskX 2837
?copy 59
?CorrExec 2851
?CorrExec1D 2855
?CorrExecX 2858
?CorrExecX1D 2862
?CorrNewTask 2833
?CorrNewTask1D 2835
?CorrNewTaskX 2837
?CorrNewTaskX1D 2841
?dbtf2 2408
?dbtrf 2410
?disna 973
?dot 60
?dotc 63
?dotci 172
?doti 171
?dotu 64
?dotui 173
?dtsvb 707
?dttrf 2412
?dttrfb 428
?dttrsrb 466
?dttrsv 2413
?feast_hbev 2562
?feast_hbgv 2565
?feast_hcsrev 2568
?feast_hcsrv 2571
?feast_heev 2556
?feast_hegv 2559
?feast_sbev 2562
?feast_sbvg 2565
?feast_scsrev 2568
?feast_scsrv 2571
?feast_srci 2552
?feast_syev 2556
?feast_sygv 2559
?gamm2d 3259
?gamx2d 3258
?gbbrd 883
?gbcon 504
?gbequ 647
?gbequb 650
?gbmv 81
?gbtrfs 547
?gbtrfsx 550
?gbsv 684
?gbsvx 686
?gbsvxx 692
?gbtf2 1385
?gbtrf 424
?gbtrs 461
?gebak 1009
?gebal 1006
?gebd2 1386
?gebr2d 3269
?gebrd 879
?gebs2d 3268
?gecon 502
?geeque 643
?geequeb 645
?gees 1208
?geesx 1213
?geev 1219
?geevx 1223
?gehd2 1388
?gehrd 993
?gejsv 1238
?gelq2 1390
?gelqf 821
?gels 1104
?gelsd 1114
?gelss 1111
?gelsy 1107
?gem2vc 392
?gem2vu 390
?gemm 139
?gemm3m 395
?gemqrt 804
?gemv 84
?geql2 1391
?geqlf 833
?geqp3 809
?geqpff 806
?geqr2 1393
?geqr2p 1394
?geqrft 796
?geqrfp 799
?geqrft 802
?geqrft2 1396
?geqrft3 1398
?ger 87
?gerc 89
?gerfs 537
?gerfsx 540
?gerq2 1400
?gerqf 845
?geru 91
?gerv2d 3265
?gesc2 1401
?gesd2d 3264

?gesdd 1234	?hetrd 921
?gesv 665	?hetrf 449
?gesvd 1229	?hetrf_rook 451
?gesvj 1245	?hetri 622
?gesvx 669	?hetri_rook 624
?gesvxx 675	?hetri2 628
?getc2 1402	?hetri2x 632
?getf2 1404	?hetrs 482
?getrf 421	?hetrs_rook 484
?getri 612	?hetrs2 488
?getrs 459	?hfrk 1746
?ggbak 1048	?hgeqz 1050
?ggbal 1045	?hpcon 528
?gges 1332	?hpev 1157
?ggesx 1338	?hpevd 1163
?ggev 1345	?hpevx 1170
?ggevx 1350	?hpgst 982
?ggglm 1121	?hpgv 1291
?gghrd 1042	?hpgvd 1297
?gglse 1118	?hpgvx 1306
?ggqrft 865	?hpmv 102
?ggrqf 868	?hpr 104
?ggsvd 1249	?hpr2 106
?ggsvp 1080	?hprfs 600
?gsum2d 3261	?hpsv 788
?gsvj0 1739	?hpsvx 790
?gsvj1 1742	?hptrd 934
?gtcon 506	?hptrf 457
?gthr 175	?hptri 636
?gthrz 176	?hptrs 492
?gtrfs 557	?hsein 1016
?gtsv 701	?hseqr 1011
?gtsvx 702	?isnan 1406
?gttrf 426	?jacobi 3175
?gttrs 464	?jacobi_delete 3175
?gtts2 1405	?jacobi_init 3172
?hbev 1176	?jacobi_solve 3173
?hbevd 1182	?jacobix 3177
?hbevx 1190	?la_gbamv 1773
?hbgst 986	?la_gbrcond 1775
?hbgv 1312	?la_gbrcond_c 1777
?hbgvd 1319	?la_gbrcond_x 1779
?hbgvx 1327	?la_gbrfsx_extended 1781
?hbtrd 942	?la_gbrpvgrw 1787
?hecon 522	?la_gearmv 1788
?hecon_rook 524	?la_gercond 1790
?heequb 663	?la_gercond_c 1792
?heev 1127	?la_gercond_x 1793
?heevd 1133	?la_gerfsx_extended 1794
?heevr 1150	?la_gerpvgrw 1829
?heevx 1141	?la_heimv 1800
?heft2 1722, 1724	?la_hercond_c 1802
?hegst 977	?la_hercond_x 1804
?hegv 1268	?la_herfsx_extended 1805
?hegvd 1275	?la_herpvgrw 1811
?hegvx 1284	?la_porcond 1813
?hemm 143	?la_porcond_c 1815
?hemv 96	?la_porcond_x 1816
?her 98	?la_porfsx_extended 1818
?her2 100	?la_porpvgrw 1824
?her2k 148	?la_syamv 1737, 1832
?herdb 914	?la_syrcond 1834
?herfs 588	?la_syrcond_c 1836
?herfsx 591	?la_syrcond_x 1837
?herk 146	?la_syrsfx_extended 1839
?hesv 764	?la_syrpvgrw 1845
?hesv_rook 767	?la_wwaddw 1847
?hesvx 769	?labrd 1407
?hesvxx 774	?lacgv 1371
?heswapr 1712	?lacn2 1410

?lacon 1412
 ?lacp2 1772
 ?lacpy 1413
 ?lacrm 1371
 ?lacrt 1372
 ?ladv 1414
 ?lae2 1415
 ?laebz 1416
 ?laed0 1420
 ?laed1 1423
 ?laed2 1424
 ?laed3 1426
 ?laed4 1428
 ?laed5 1430
 ?laed6 1431
 ?laed7 1432
 ?laed8 1435
 ?laed9 1438
 ?laeda 1440
 ?laein 1442
 ?laesy 1373
 ?laev2 1444
 ?laexc 1446
 ?lag2 1447
 ?lagge 1864
 ?laghe 1865
 ?lags2 1449
 ?lagsy 1866
 ?lagtf 1452
 ?lagtm 1454
 ?lagts 1455
 ?lagv2 1457
 ?lahef 1651
 ?lahef_rook 1653
 ?lahqr 1459
 ?lahr2 1464
 ?lahrd 1461
 ?laic1 1466
 ?laisnan 1407
 ?lain2 1468
 ?lals0 1470
 ?lalsa 1474
 ?lalsd 1477
 ?lamc1 1856
 ?lamc2 1856
 ?lamc3 1857
 ?lamc4 1858
 ?lamc5 1859
 ?lamch 1855
 ?lamrg 1479
 ?lamsh 2361
 ?laneg 1480
 ?langb 1481
 ?lange 1483
 ?langt 1484
 ?lanhb 1488
 ?lanhe 1495
 ?lanhf 1752
 ?lanhp 1491
 ?lanhs 1485
 ?lansb 1486
 ?lansf 1750
 ?lansp 1489
 ?lanst/?lanht 1492
 ?lansy 1493
 ?lantb 1496
 ?lantp 1498
 ?lantr 1500
 ?lanv2 1502
 ?lapll 1503
 ?lapmr 1504
 ?lapmt 1505
 ?lapy2 1506
 ?lapy3 1507
 ?laqgb 1508
 ?laqge 1509
 ?laqhb 1511
 ?laqhe 1825
 ?laqhp 1827
 ?laqp2 1513
 ?laqps 1514
 ?laqr0 1516
 ?laqr1 1519
 ?laqr2 1520
 ?laqr3 1524
 ?laqr4 1527
 ?laqr5 1531
 ?laqsb 1534
 ?laqsp 1535
 ?laqsy 1537
 ?laqtr 1538
 ?lar1v 1541
 ?lar2v 1544
 ?larcm 1828
 ?laref 2369
 ?larf 1545
 ?larfb 1547
 ?larfg 1550
 ?larfgp 1551
 ?larfp 1736
 ?larft 1553
 ?larfx 1555
 ?largv 1557
 ?larnv 1558
 ?larra 1560
 ?larrb 1561
 ?larrc 1563
 ?larrd 1564
 ?larre 1568
 ?larrf 1571
 ?larrj 1573
 ?larrk 1575
 ?larrr 1577
 ?larrv 1578
 ?larscl2 1830
 ?lartg 1581
 ?lartgp 1583
 ?lartgs 1584
 ?lartv 1586
 ?laruv 1587
 ?larz 1588
 ?larzb 1590
 ?larzt 1592
 ?las2 1595
 ?lascl 1595
 ?lascl2 1831
 ?lasd0 1597
 ?lasd1 1598
 ?lasd2 1601
 ?lasd3 1604
 ?lasd4 1607
 ?lasd5 1608
 ?lasd6 1609
 ?lasd7 1613
 ?lasd8 1617
 ?lasd9 1619
 ?lasda 1621
 ?lasdq 1624
 ?lasdt 1627
 ?laset 1627

?lasorte 2393
?lasq1 1629
?lasq2 1630
?lasq3 1631
?lasq4 1633
?lasq5 1634
?lasq6 1636
?lasr 1637
?lasrt 1640
?lasrt2 2394
?lassq 1641
?lasv2 1642
?laswp 1643
?lasy2 1644
?lasyf 1647
?lasyf_rook 1649
?latbs 1655
?latdf 1657
?latms 1867
?latps 1659
?latrd 1661
?latrs 1663
?latrz 1667
?lauu2 1669
?lauum 1670
?nrm2 65
?opgr 930
?opmtr 932
?orbdb/?unbdb 1098
?orbdb1/?unbdb1 1671
?orbdb2/?unbdb2 1674
?orbdb3/?unbdb3 1678
?orbdb4/?unbdb4 1681
?orbdb5/?unbdb5 1684
?orbdb6/?unbdb6 1687
?orcisd/?uncsd 1255
?orcisd2by1/?orcisd2by1 1261
?org2l/?ung2l 1689
?org2r/?ung2r 1691
?orgbr 886
?orghr 996
?orgl2/?ungl2 1692
?orglq 824
?orgql 836
?orgqr 812
?orgr2/?ungr2 1694
?orgrq 848
?orgtr 916
?orm2l/?unm2l 1695
?orm2r/?unm2r 1697
?ormbr 889
?ormhr 998
?orml2/?unml2 1699
?ormlq 826
?ormql 840
?ormqr 814
?ormr2/?unmr2 1701
?ormr3/?unmr3 1703
?ormrq 851
?ormrz 859
?ormtr 918
?pbcon 513
?pbequ 658
?pbrfs 573
?pbstf 989
?pbsv 733
?pbsvx 735
?pbtf2 1706
?pbtrf 438
?pbtrs 474
?pftrf 434
?pftri 615
?pftrs 470
?pocon 509
?poequ 652
?poequb 654
?porfs 560
?porfsx 563
?posv 709
?posvx 713
?posvxx 718
?potf2 1707
?potrf 430
?potri 614
?potrs 468
?ppcon 511
?ppequ 656
?pprfs 570
?ppsv 726
?ppsvx 728
?pptrf 436
?pptri 617
?pptrs 472
?pstf2 1768
?pstrf 432
?ptcon 516
?pteqr 964
?ptrfs 576
?pts 740
?ptsx 742
?pttrf 440
?pttrs 476
?pttvs 2415
?ptts2 1708
?rot 67, 1375
?rotg 68
?roti 177
?rotm 70
?rotmg 72
?rscl 1710
?sbev 1174
?sbevd 1179
?sbevx 1186
?sbgst 984
?sbgv 1310
?sbgvd 1315
?sbgvx 1323
?sbmv 108
?sbtrd 940
?scal 74
?scdr 179
?sdot 61
?sfrk 1744
?spcon 526
?spev 1155
?spevd 1159
?spevx 1166
?spgst 980
?spgv 1289
?spgvd 1294
?spgvx 1301
?spmv 111, 1376
?spr 113, 1377
?spr2 115
?sprfs 597
?spsv 781
?spsvx 783
?sptrd 928
?sptrf 454
?sptri 634

?sptrs 490
?stebz 967
?stedc 954
?steqr 959
?stein 970
?stemr 950
?steqr 946
?steqr2 2416
?sterf 945
?stev 1194
?stevd 1196
?stevr 1203
?stevx 1200
?sum1 1384
?swap 75
?sycl 518
?sycl_rook 520
?syclv 1379
?syequb 661
?syev 1125
?syevd 1130
?syevr 1145
?syevx 1137
?sygs2/?hegs2 1715
?sygst 975
?sygv 1265
?sygvd 1271
?sygvx 1279
?somm 151
?symv 117, 1380
?syr 119, 1382
?syr2 121
?syr2k 158
?syrdb 912
?syrfs 579
?syrfsx 582
?syrk 155
?sylv 746
?sylv_rook 749
?sylvx 751
?sylvxx 756
?syswapr 1711
?syswapr1 1714
?sytd2/?hetd2 1717
?sytf2 1719
?sytf2_rook 1721
?sytrd 909
?sytrf 441
?sytrf_rook 445
?sytri 619
?sytri_rook 620
?sytri2 626
?sytri2x 630
?syttrs 478
?sytrs_rook 480
?sytrs2 486
?tbcn 534
?tbtmv 123
?tbsv 126
?tbttrs 499
?tfsm 1748
?tftri 639
?tftrp 1753
?tftrr 1754
?tgevc 1057
?tgex2 1725
?tgexc 1061
?tgsen 1064
?tgsja 1084
?tgsna 1075
?tgsy2 1727
?tgsyl 1070
?tbcn 532
?tpmqrt 874
?tpmv 129
?tpqrt 872
?tpqrt2 1756
?tprb 1759
?tprfs 605
?tpsv 131
?tptri 641
?tptrs 497
?tpttf 1762
?tptrr 1764
?trbr2d 3270
?trbs2d 3268
?trcon 530
?trevc 1021
?trexc 1031
?trmm 161
?trmv 134
?trnlsp_check 3157
?trnlsp_delete 3162
?trnlsp_get 3161
?trnlsp_init 3155
?trnlsp_solve 3159
?trnlspbc_check 3165
?trnlspbc_delete 3171
?trnlspbc_get 3170
?trnlspbc_init 3163
?trnlspbc_solve 3168
?trrfs 603
?trrv2d 3266
?trsd2d 3264
?trsen 1033
?trsm 164
?trsna 1026
?trsv 136
?trsvl 1038
?trti2 1731
?trtri (LAPACK) 638
?trtrs (LAPACK) 494
?trtf 1765
?trtp 1767
?tzrzf 856
?ungbr 892
?unghr 1001
?ungle 829
?ungql 838
?ungqr 816
?ungrq 849
?ungtr 924
?unmbr 895
?unmhr 1003
?unmlq 831
?unmql 842
?unmqr 819
?unmrq 854
?unmrz 862
?unmtr 926
?upgr 936
?upmtr 938

1-norm value
complex Hermitian matrix
packed storage 1491
complex Hermitian matrix in RFP format 1752
complex Hermitian tridiagonal matrix 1492
complex symmetric matrix 1493

general rectangular matrix 1483, 2219
 general tridiagonal matrix 1484
 Hermitian band matrix 1488
 real symmetric matrix 1493, 2223
 real symmetric matrix in RFP format 1750
 real symmetric tridiagonal matrix 1492
 symmetric band matrix 1486
 symmetric matrix
 packed storage 1489
 trapezoidal matrix 1500
 triangular band matrix 1496
 triangular matrix
 packed storage 1498
 upper Hessenberg matrix 1485, 2221

A

absolute value of a vector element
 largest 77
 smallest 78
 accuracy modes, in VML 2575
 adding magnitudes of elements of a distributed vector
 3007
 adding magnitudes of the vector elements 56
 arguments
 matrix 3366
 sparse vector 167
 vector 3365
 array descriptor 1873, 3003
 auxiliary functions
 ?la_lin_berr 1812
 auxiliary routines
 ScaLAPACK 2155

B

backward error 1812
 balancing a matrix 1006
 band storage scheme 3366
 basic quasi-number generator
 Niederreiter 2732
 Sobol 2732
 basic random number generators
 GFSR 2732
 MCG, 32-bit 2732
 MCG, 59-bit 2732
 Mersenne Twister
 MT19937 2732
 MT2203 2732
 MRG 2732
 Wichmann-Hill 2732
 bdlaapp 2160
 bdlaexc 2162
 bdsdc 902
 bdtrexc 2163
 Bernoulli 2808
 Beta 2798
 bidiagonal matrix
 LAPACK 878
 ScaLAPACK 2050
 Binomial 2811
 bisection 1561
 BLACS
 broadcast 3266
 combines 3257
 destruction routines 3277
 informational routines 3279
 initialization routines 3271
 miscellaneous routines 3281

point to point communication 3262
 ?gamn2d 3259
 ?gamx2d 3258
 ?gebr2d 3269
 ?gebs2d 3268
 ?gerv2d 3265
 ?gesd2d 3264
 ?gsum2d 3261
 ?trbr2d 3270
 ?trbs2d 3268
 ?trrv2d 3266
 ?trsd2d 3264
 blacs_abort 3278
 blacs_barrier 3281
 blacs_exit 3279
 blacs_freebuff 3278
 blacs_get 3272
 blacs_gridexit 3278
 blacs_gridinfo 3280
 blacs_gridinit 3275
 blacs_gridmap 3276
 blacs_pcoord 3280
 blacs_pinfo 3271
 blacs_pnum 3280
 blacs_set 3273
 blacs_setup 3272
 usage examples 3282
 BLACS routines
 coherence 3254
 matrix shapes 3253
 repeatability 3254
 blacs_abort 3278
 blacs_barrier 3281
 blacs_exit 3279
 blacs_freebuff 3278
 blacs_get 3272
 blacs_gridexit 3278
 blacs_gridinfo 3280
 blacs_gridinit 3275
 blacs_gridmap 3276
 blacs_pcoord 3280
 blacs_pinfo 3271
 blacs_pnum 3280
 blacs_set 3273
 blacs_setup 3272
 BLAS Code Examples 3395
 BLAS Level 1 routines
 ?asum 55, 56
 ?axpby 388
 ?axpy 55, 57
 ?cabs1 55, 79
 ?copy 55, 59
 ?dot 55, 60
 ?dotc 55, 63
 ?dotu 55, 64
 ?nrm2 55, 65
 ?rot 55, 67
 ?rotg 55, 68
 ?rotm 55, 70
 ?rotmg 72
 ?rotmq 55
 ?scal 55, 74
 ?sdot 55, 61
 ?swap 55, 75
 code example 3395
 i?amax 55, 77
 i?amin 55, 78
 BLAS Level 2 routines
 ?gbmv 80, 81
 ?gem2vc 392

?gem2vu 390
?gemv 80, 84
?ger 80, 87
?gerc 80, 89
?geru 80, 91
?hbmv 80, 93
?hemv 80, 96
?her 80, 98
?her2 80, 100
?hpmv 80, 102
?hpr 80, 104
?hpr2 80, 106
?sbbmv 80, 108
?spmv 80, 111
?spr 80, 113
?spr2 80, 115
?symv 80, 117
?syr 80, 119
?syr2 80, 121
?tbbmv 80, 123
?tbsv 80, 126
?tpmv 80, 129
?tpsv 80, 131
?trmv 80, 134
?trsv 80, 136
code example 3396

BLAS Level 3 routines
?gemm 138, 139
?gemm3m 395
?hemm 138, 143
?her2k 138, 148
?herk 138, 146
?symm 138, 151
?syr2k 138, 158
?syrk 138, 155
?tfsrm 1748
?trmm 138, 161
?trsm 138, 164
code example 3397

BLAS routines
routine groups 51

BLAS-like extensions 387

BLAS-like transposition routines
mkl_?imatcopy 398
mkl_?omatadd 407
mkl_?omatcopy 401
mkl_?omatcopy2 404

block reflector
general matrix
LAPACK 1590
ScalAPACK 2273
general rectangular matrix
LAPACK 1547
ScalAPACK 2257
triangular factor
LAPACK 1553, 1592
ScalAPACK 2266, 2280

block-cyclic distribution 1873, 3003

block-splitting method 2732

BRNG 2725, 2726, 2732

bslaapp 2160
bslaexc 2162
bstrexc 2163

Bunch-Kaufman factorization
Hermitian matrix
packed storage 457
symmetric matrix
packed storage 454

C

C interface conventions
BLAS, Sparse BLAS 53
LAPACK 412
Cauchy 2785
cbbcsd 1091
CBLAS
arguments 54
CBLAS to the BLAS 53
cgbcon 504
cgbfsx 550
cgbsvx 686
cgbtrs 461
cgecon 502
cgeqpf 806
cgttrs 557
chegs2 1715
cheswapr 1712
chetd2 1717
chetri2 628
chetri2x 632
chetrs2 488
chgeqz 1050
chla_transtype 1859
Cholesky factorization
Hermitian positive semi-definite matrix 1768
Hermitian positive semidefinite matrix 432
Hermitian positive-definite matrix
band storage 438, 474, 735, 1888, 1905
packed storage 436, 728
split 989
symmetric positive semi-definite matrix 1768
symmetric positive semidefinite matrix 432
symmetric positive-definite matrix
band storage 438, 474, 735, 1888, 1905
packed storage 436, 728

chseqr 1011
cla_gbamv 1773
cla_gbrcond_c 1777
cla_gbrcond_x 1779
cla_gbrfsx_extended 1781
cla_gbrpvgrw 1787
cla_geamv 1788
cla_gercond_c 1792
cla_gercond_x 1793
cla_gerfsx_extended 1794
cla_gerpvgrw 1829
cla_heamv 1800
cla_hercond_c 1802
cla_hercond_x 1804
cla_herfsx_extended 1805
cla_herpvggrw 1811
cla_linserr 1812
cla_porcond_c 1815
cla_porcond_x 1816
cla_porfsx_extended 1818
cla_porpvggrw 1824
cla_syamv 1832
cla_syrcond_c 1836
cla_syrcond_x 1837
cla_syrfsx_extended 1839
cla_syrpvgrw 1845
cla_wwaddw 1847
clag2z 1732
clapmr 1504
clapmt 1505
clarfb 1547
clarft 1553
clarscl2 1830
clascl2 1831
clatps 1659

clatrd 1661
 clatrs 1663
 clatrz 1667
 clauu2 1669
 clauum 1670
 cluster_sparse_solver 2469
 cluster_sparse_solver iparm parameter 2474
 CNR, support functions for
 mkl_cbwr_get 3241
 mkl_cbwr_get_auto_branch 3242
 mkl_cbwr_set 3240
 code examples
 BLAS Level 1 function 3395
 BLAS Level 1 routine 3395
 BLAS Level 2 routine 3396
 BLAS Level 3 routine 3397
 communication subprograms 1873
 complex division in real arithmetic 1414
 complex Hermitian matrix
 1-norm value
 LAPACK 1495
 ScalAPACK 2223
 factorization with diagonal pivoting method 1722,
 1724
 Frobenius norm
 LAPACK 1495
 ScalAPACK 2223
 infinity- norm
 LAPACK 1495
 ScalAPACK 2223
 largest absolute value of element
 LAPACK 1495
 ScalAPACK 2223
 complex Hermitian matrix in packed form
 1-norm value 1491
 Frobenius norm 1491
 infinity- norm 1491
 largest absolute value of element 1491
 complex Hermitian tridiagonal matrix
 1-norm value 1492
 Frobenius norm 1492
 infinity- norm 1492
 largest absolute value of element 1492
 complex matrix
 complex elementary reflector
 ScalAPACK 2277
 complex symmetric matrix
 1-norm value 1493
 Frobenius norm 1493
 infinity- norm 1493
 largest absolute value of element 1493
 complex vector
 1-norm using true absolute value
 LAPACK 1384
 ScalAPACK 2171
 conjugation
 LAPACK 1371
 ScalAPACK 2166
 complex vector conjugation
 LAPACK 1371
 ScalAPACK 2166
 component-wise relative error 1812
 compressed sparse vectors 167
 computational node 2727
 Computational Routines 794
 condition number
 band matrix 504
 general matrix
 LAPACK 502
 ScalAPACK 1912, 1915, 1918
 Hermitian matrix
 packed storage 528
 Hermitian positive-definite matrix
 band storage 513
 packed storage 511
 tridiagonal 516
 symmetric matrix
 packed storage 526
 symmetric positive-definite matrix
 band storage 513
 packed storage 511
 tridiagonal 516
 triangular matrix
 band storage 534
 packed storage 532
 tridiagonal matrix 506
 configuration parameters, in FFT interface 2933
 Configuration Settings, for Fourier transform functions
 2934, 2977, 2980
 Continuous Distribution Generators 2764
 Continuous Distributions 2767
 ConvCopyTask 2867
 ConvDeleteTask 2866
 converting a DOUBLE COMPLEX triangular matrix to
 COMPLEX 1771
 converting a double-precision triangular matrix to single-
 precision 1770
 converting a sparse vector into compressed storage form
 and writing zeros to the original vector 176
 converting compressed sparse vectors into full storage
 form 179
 ConvInternalPrecision 2847
 Convolution and Correlation 2827
 Convolution Functions
 ?ConvExec 2851
 ?ConvExec1D 2855
 ?ConvExecX 2858
 ?ConvExecX1D 2862
 ?ConvNewTask 2833
 ?ConvNewTask1D 2835
 ?ConvNewTaskX 2837
 ?ConvNewTaskX1D 2841
 ConvCopyTask 2867
 ConvDeleteTask 2866
 ConvSetDecimation 2849
 ConvSetInternalPrecision 2847
 ConvSetMode 2845
 ConvSetStart 2848
 CorrCopyTask 2867
 CorrDeleteTask 2866
 ConvSetMode 2845
 ConvSetStart 2848
 copying
 distributed vectors 3010
 matrices
 distributed 2206
 global parallel 2208
 local replicated 2208
 two-dimensional
 LAPACK 1413, 1772
 ScalAPACK 2209
 vectors 59
 copying a matrix 1753, 1754, 1762, 1764, 1765, 1767
 CopyStream 2749
 CopyStreamState 2750
 CorrCopyTask 2867
 CorrDeleteTask 2866
 Correlation Functions
 ?CorrExec 2851
 ?CorrExec1D 2855

?CorrExecX 2858
?CorrExecX1D 2862
?CorrNewTask 2833
?CorrNewTask1D 2835
?CorrNewTaskX 2837
?CorrNewTaskX1D 2841
CorrSetDecimation 2849
CorrSetInternalPrecision 2847
CorrSetMode 2845
CorrSetStart 2848
CorrSetInternalDecimation 2849
CorrSetInternalPrecision 2847
CorrSetMode 2845
CorrSetStart 2848
cosine-sine decomposition
 LAPACK 1091, 1255
cpbtf2 1706
cporfsx 563
cpotf2 1707
cpprfs 570
cpptrs 472
cptts2 1708
Cray 2419
crscl 1710
cs decomposition
 See also LAPACK routines, cs decomposition 1091
CSD (cosine-sine decomposition)
 LAPACK 1091, 1255
csyconv 1379
csyswapr 1711
csyswapr1 1714
csytf2 1719
csytf2_rook 1721
csytri2 626
csytri2x 630
csytrs2 486
ctgex2 1725
ctgsy2 1727
ctrexc 1031
ctrri2 1731
cunbdb 1098
cunbdb1 1671
cunbdb2 1674
cunbdb3 1678
cunbdb4 1681
cunbdb5 1684
cunbdb6 1687
cuncsd 1255
cuncsd2by1 1261
cung2l 1689
cung2r 1691
cungbr 892
cungl2 1692
cungr2 1694
cunm2l 1695
cunm2r 1697
cunml2 1699
cunmr2 1701
cunmr3 1703

D

data type
 in VML 2575
 shorthand 41
Data Types 2735
Datatypes, C language 49
dbbscd 1091
dbdsdc 902
dcg_check 2512

dcg_get 2515
dcg_init 2511
dcgmrhs_check 2517
dcgmrhs_get 2520
dcgmrhs_init 2516
DeleteStream 2749
descriptor configuration
 cluster FFT 2989
descriptor manipulation
 cluster FFT 2989
DF task 3293
dfdconstruct1d 3324
dfdConstruct1D 3324
dfdeditidxptr 3318
dfdEditIdxPtr 3318
dfdeditppspline1d 3308
dfdEditPPSpline1D 3308
dfdeditptr 3315
dfdEditPtr 3315
dfdeletetask 3345
dfDeleteTask 3345
dfdintegrate1d 3332
dfdIntegrate1D 3332
dfdintegrateex1d 3332
dfdIntegrateEx1D 3332
dfdintegrcallback 3341
dfdIntegrCallBack 3341
dfdinterpcallback 3340
dfdInterpCallBack 3340
dfdinterpolate1d 3325
dfdInterpolate1D 3325
dfdinterpolateex1d 3325
dfdInterpolateEx1D 3325
dfdnewtask1d 3305
dfdNewTask1D 3305
dfdqueryidxptr 3322
dfdQueryIdxPtr 3322
dfdqueryptr 3320
dfdQueryPtr 3320
dfdsearchcells1d 3337
dfdSearchCells1D 3337
dfdsearchcellcallback 3343
dfdSearchCellsCallBack 3343
dfdsearchcellsex1d 3337
dfdSearchCellsEx1D 3337
dfgmres_check 2522
dfgmres_get 2525
dfgmres_init 2521
dfieditval 3316
dfiEditVal 3316
dfiqueryval 3321
dfiQueryVal 3321
dfsconstruct1d 3324
dfsConstruct1D 3324
dfseditidxptr 3318
dfsEditIdxPtr 3318
dfseditppspline1d 3308
dfsEditPPSpline1D 3308
dfseditptr 3315
dfsEditPtr 3315
dfsintegrate1d 3332
dfsIntegrate1D 3332
dfsintegrateex1d 3332
dfsIntegrateEx1D 3332
dfsintegrcallback 3341
dfsIntegrCallBack 3341
dfsinterpcallback 3340
dfsInterpCallBack 3340
dfsinterpolate1d 3325
dfsInterpolate1D 3325

dfsinterpolateex1d 3325
 dfsInterpolateEx1D 3325
 dfsnewtask1d 3305
 dfsNewTask1D 3305
 dfsqueryidxptr 3322
 dfsQueryIdxPtr 3322
 dfsqueryptr 3320
 dfsQueryPtr 3320
 dfssearchcells1d 3337
 dfsSearchCells1D 3337
 dfssearchcellcallback 3343
 dfsSearchCellsCallBack 3343
 dfssearchcellsex1d 3337
 dfsSearchCellsEx1D 3337
 DFT routines
 descriptor configuration
 DftiSetValue 2967
 DftiCommitDescriptor 2963
 DftiCommitDescriptorDM 2991
 DftiComputeBackward 2974
 DftiComputeBackwardDM 2995
 DftiComputeForward 2971
 DftiComputeForwardDM 2993
 DftiCopyDescriptor 2966
 DftiCreateDescriptor 2960
 DftiCreateDescriptorDM 2990
 DftiErrorClass 2983
 DftiErrorMessage 2984
 DftiFreeDescriptor 2964
 DftiFreeDescriptorDM 2992
 DftiGetValue 2969
 DftiGetValueDM 2999
 DftiSetValue 2967
 DftiSetValueDM 2997
 dgbcon 504
 dgbrfsx 550
 dgbsvx 686
 dgbrtrs 461
 dgecon 502
 dgejsv 1238
 dgeqpf 806
 dgesvj 1245
 dgtrfs 557
 dhgeqz 1050
 dhseqr 1011
 diagonal elements
 LAPACK 1627
 ScaLAPACK 2285
 diagonal pivoting factorization
 Hermitian indefinite matrix 774
 symmetric indefinite matrix 756
 diagonally dominant tridiagonal matrix
 solving systems of linear equations 466
 diagonally dominant-like banded matrix
 solving systems of linear equations 1898
 diagonally dominant-like tridiagonal matrix
 solving systems of linear equations 1900
 dimension 3365
 Direct Sparse Solver (DSS) Interface Routines 2478
 Discrete Distribution Generators 2764, 2765
 Discrete Distributions 2801
 Discrete Fourier Transform
 DftiSetValue 2967
 distributed complex matrix
 transposition 3083, 3085
 distributed general matrix
 matrix-vector product 3022, 3025
 rank-1 update 3028
 rank-1 update, unconjugated 3032
 rank-1 update, conjugated 3030
 distributed Hermitian matrix
 matrix-vector product 3034, 3036
 rank-1 update 3038
 rank-2 update 3040
 rank-k update 3069
 distributed matrix equation
 $AX = B$ 3089
 distributed matrix-matrix operation
 rank-k update
 distributed Hermitian matrix 3069
 transposition
 complex matrix 3083
 complex matrix, conjugated 3085
 real matrix 3082
 distributed matrix-vector operation
 product
 Hermitian matrix 3034, 3036
 symmetric matrix 3042, 3044
 triangular matrix 3051, 3053
 rank-1 update
 Hermitian matrix 3038
 symmetric matrix 3047
 rank-1 update, conjugated 3030
 rank-1 update, unconjugated 3032
 rank-2 update
 Hermitian matrix 3040
 symmetric matrix 3048
 distributed real matrix
 transposition 3082
 distributed symmetric matrix
 matrix-vector product 3042, 3044
 rank-1 update 3047
 rank-2 update 3048
 distributed triangular matrix
 matrix-vector product 3051, 3053
 solving systems of linear equations 3056
 distributed vector-scalar product 3018
 distributed vectors
 adding magnitudes of vector elements 3007
 copying 3010
 dot product
 complex vectors 3015
 complex vectors, conjugated 3014
 real vectors 3012
 Euclidean norm 3017
 global index of maximum element 3006
 linear combination of vectors 3009
 sum of vectors 3009
 swapping 3019
 vector-scalar product 3018
 distributed-memory computations 1873
 Distribution Generators 2764
 Distribution Generators Supporting Accurate Mode 2765
 divide and conquer algorithm 2103, 2120
 djacobi 3175
 djacobi_delete 3175
 djacobi_init 3172
 djacobi_solve 3173
 djacobix 3177
 dla_gbamv 1773
 dla_gbrcond 1775
 dla_gbtrfsx_extended 1781
 dla_gbrpvgrw 1787
 dla_geamv 1788
 dla_gercond 1790
 dla_gerfsx_extended 1794
 dla_gerpvgrw 1829
 dla_lin_berr 1812
 dla_porcond 1813
 dla_porfsx_extended 1818

dla_porpvgrw 1824
 dla_syamv 1832
 dla_syrcond 1834
 dla_syrfsx_extended 1839
 dla_syrvgrw 1845
 dla_wwaddw 1847
 dlag2s 1733
 dlapmr 1504
 dlapmt 1505
 dlaqr6 2363
 dlar1va 2366
 dlarfb 1547
 dlarft 1553
 dlarrb2 2371
 dlarrd2 2374
 dlarre2 2377
 dlarre2a 2381
 dlarrf2 2386
 dlarrv2 2388
 dlarscl2 1830
 dlartgp 1583
 dlartgs 1584
 dlascl2 1831
 dlat2s 1770
 dlatps 1659
 dlatrd 1661
 dlatrs 1663
 dlatrz 1667
 dlauu2 1669
 dlauum 1670
 dNewAbstractStream 2744
 dorbdb 1098
 dorbdb1 1671
 dorbdb2 1674
 dorbdb3 1678
 dorbdb4 1681
 dorbdb5 1684
 dorbdb6 1687
 dorcsd 1255
 dorcsd2by1 1261
 dorg2l 1689
 dorg2r 1691
 dorgl2 1692
 dorgr2 1694
 dorm2l 1695
 dorm2r 1697
 dorml2 1699
 dormr2 1701
 dormr3 1703
 dot product

- complex vectors, conjugated 63
- complex vectors, unconjugated 64
- distributed complex vectors, conjugated 3014
- distributed complex vectors, unconjugated 3015
- distributed real vectors 3012
- real vectors 60
- real vectors (double precision) 61
- sparse complex vectors 173
- sparse complex vectors, conjugated 172
- sparse real vectors 171

dpbtf2 1706
 dporfsx 563
 dptf2 1707
 dpprfs 570
 dptrs 472
 dpts2 1708
 driver

- expert 1875
- simple 1875

Driver Routines 664, 1103

drscl 1710
 dss_create 2482
 dstegr2 2395
 dstegr2a 2399
 dstegr2b 2402
 dsyconv 1379
 dsygs2 1715
 dsyswapr 1711
 dsyswapr1 1714
 dsytd2 1717
 dsytf2 1719
 dsytf2_rook 1721
 dsytri2 626
 dsytri2x 630
 dsytrs2 486
 dtgex2 1725
 dtgsy2 1727
 dtrexc 1031
 dtrnlsp_check 3157
 dtrnlsp_delete 3162
 dtrnlsp_get 3161
 dtrnlsp_init 3155
 dtrnlsp_solve 3159
 dtrnlspbc_check 3165
 dtrnlspbc_delete 3171
 dtrnlspbc_get 3170
 dtrnlspbc_init 3163
 dtrnlspbc_solve 3168
 dtrti2 1731
 dzsum1 1384

E

eigenpairs, sorting 2393
 eigenvalue problems

- general matrix 990, 1041, 2035
- generalized form 975
- Hermitian matrix 905
- symmetric matrix 905
- symmetric tridiagonal matrix 2406, 2416

eigenvalues

- bisection 2371
- eigenvalue problems 905

eigenvalues and eigenvectors 2399

eigenvectors

- eigenvalue problems 905
- scaled 2366

elementary reflector

- complex matrix 2277
- general matrix 1588, 2269
 - general rectangular matrix
 - LAPACK 1545, 1555
 - ScalAPACK 2254, 2261
- LAPACK generation 1550, 1551
- ScalAPACK generation 2264

error diagnostics, in VML 2580
 error estimation for linear equations

- distributed tridiagonal coefficient matrix 1928

error handling

- pxerbla 3199
- xerbla 51, 2580, 3003

error handling for fatal errors 3200

errors in solutions of linear equations

- banded matrix 550, 1781, 1818
- distributed tridiagonal coefficient matrix 1928
 - general matrix
 - band storage 547
- Hermitian indefinite matrix 591, 1805
- Hermitian matrix

packed storage 600
 Hermitian positive-definite matrix
 band storage 573
 packed storage 570
 symmetric indefinite matrix 582, 1839
 symmetric matrix
 packed storage 597
 symmetric positive-definite matrix
 band storage 573
 packed storage 570
 triangular matrix
 band storage 608
 packed storage 605
 tridiagonal matrix 557
 Estimates 3323
 Euclidean norm
 of a distributed vector 3017
 of a vector 65
 exception handling 3198
 expert driver 1875
 Exponential 2777
 Extended Eigensolver 2543
 Extended Eigensolver interface
 ?feast_hbev 2562
 ?feast_hbgv 2565
 ?feast_hcsrev 2568
 ?feast_hcsgrv 2571
 ?feast_heev 2556
 ?feast_hegv 2559
 ?feast_hrci 2552
 ?feast_sbev 2562
 ?feast_sbvg 2565
 ?feast_scsrev 2568
 ?feast_scsgrv 2571
 ?feast_srci 2552
 ?feast_syev 2556
 ?feast_sygv 2559
 feastinit 2547
 Extended Eigensolvers 2545–2550, 2552, 2556, 2559,
 2562, 2565, 2568, 2571

F

factorization
 Bunch-Kaufman
 LAPACK 421
 ScalAPACK 1877
 Cholesky
 LAPACK 421, 1706, 1707
 ScalAPACK 2340
 diagonal pivoting
 Hermitian matrix
 complex 1722, 1724
 packed 790
 symmetric matrix
 indefinite 1719, 1721
 packed 783
 LU
 LAPACK 421
 ScalAPACK 1877
 orthogonal
 LAPACK 795
 ScalAPACK 1943
 partial
 complex Hermitian indefinite matrix 1651,
 1653
 real/complex symmetric matrix 1647, 1649
 triangular factorization 421, 1877
 upper trapezoidal matrix 1667

fast Fourier transform
 DftiCommitDescriptor 2963
 DftiCommitDescriptorDM 2991
 DftiComputeBackward 2974
 DftiComputeBackwardDM 2995
 DftiComputeForwardDM 2993
 DftiCopyDescriptor 2966
 DftiCreateDescriptor 2960
 DftiCreateDescriptorDM 2990
 DftiErrorClass 2983
 DftiErrorMessage 2984
 DftiFreeDescriptor 2964
 DftiFreeDescriptorDM 2992
 DftiGetValue 2969
 DftiGetValueDM 2999
 DftiSetValueDM 2997
 fast Fourier Transform
 DftiComputeForward 2971
 fatal error, handling 3200
 feastinit 2547
 FFT computation
 cluster FFT 2989
 FFT functions
 descriptor manipulation
 DftiCommitDescriptor 2963
 DftiCommitDescriptorDM 2991
 DftiCopyDescriptor 2966
 DftiCreateDescriptor 2960
 DftiCreateDescriptorDM 2990
 DftiFreeDescriptor 2964
 DftiFreeDescriptorDM 2992
 DFT computation
 DftiComputeBackward 2974
 DftiComputeForward 2971
 FFT computation
 DftiComputeForwardDM 2993
 status checking
 DftiErrorClass 2983
 DftiErrorMessage 2984
 FFT Interface 2933
 FFT routines
 descriptor configuration
 DftiGetValue 2969
 DftiGetValueDM 2999
 DftiSetValueDM 2997
 FFT computation
 DftiComputeBackwardDM 2995
 FFTW interface to Intel(R) MKL
 for FFTW2 3381
 for FFTW3 3389
 fill-in, for sparse matrices 3349
 finding
 index of the element of a vector with the largest
 absolute value of the real part 2167
 element of a vector with the largest absolute value 77
 element of a vector with the largest absolute value of
 the real part and its global index 2170
 element of a vector with the smallest absolute value
 78
 font conventions 41
 Fortran 95 interface conventions
 BLAS, Sparse BLAS 53
 LAPACK 415
 Fortran 95 Interfaces for LAPACK
 absent from Netlib 3377
 identical to Netlib 3373
 modified Netlib interfaces 3376
 new functionality 3380
 with replaced Netlib argument names 3374
 Fortran 95 Interfaces for LAPACK Routines

- specific MKL features
 - Fortran 95 LAPACK interface vs. Netlib 416
 - free_Helmholtz_2D 3128
 - free_Helmholtz_3D 3128
 - free_sph_np 3134
 - free_sph_p 3134
 - free_trig_transform 3104
 - Frobenius norm
 - complex Hermitian matrix
 - packed storage 1491
 - complex Hermitian matrix in RFP format 1752
 - complex Hermitian tridiagonal matrix 1492
 - complex symmetric matrix 1493
 - general rectangular matrix 1483, 2219
 - general tridiagonal matrix 1484
 - Hermitian band matrix 1488
 - real symmetric matrix 1493, 2223
 - real symmetric matrix in RFP format 1750
 - real symmetric tridiagonal matrix 1492
 - symmetric band matrix 1486
 - symmetric matrix
 - packed storage 1489
 - trapezoidal matrix 1500
 - triangular band matrix 1496
 - triangular matrix
 - packed storage 1498
 - upper Hessenberg matrix 1485, 2221
 - full storage scheme 3366
 - full-storage vectors 167
 - function name conventions, in VML 2576
- G**
- Gamma 2796
 - gathering sparse vector's elements into compressed form and writing zeros to these elements 176
 - Gaussian 2770
 - GaussianMV 2772
 - gbcon 504
 - gbsvx 686
 - gbtrs 461
 - gecon 502
 - general distributed matrix
 - scalar-matrix-matrix product 3063
 - general matrix
 - block reflector 1590, 2273
 - copying 2423
 - eigenvalue problems 990, 1041, 2035
 - elementary reflector 1588, 2269
 - estimating the condition number
 - band storage 504
 - inverting matrix
 - LAPACK 612
 - ScalAPACK 1932
 - LQ factorization 821, 1958
 - LU factorization
 - band storage 424, 1385, 1879, 1882, 2408, 2410
 - matrix-vector product
 - band storage 81
 - multiplying by orthogonal matrix
 - from LQ factorization 1699, 2324
 - from QR factorization 1697, 2320
 - from RQ factorization 1701, 2328
 - from RZ factorization 1703
 - multiplying by unitary matrix
 - from LQ factorization 1699, 2324
 - from QR factorization 1697, 2320
 - from RQ factorization 1701, 2328
 - LQ factorization 821, 1958
 - LU factorization
 - band storage 424, 1385, 1879, 1882, 2408, 2410
 - matrix-vector product
 - band storage 81
 - multiplying by orthogonal matrix
 - from LQ factorization 1699, 2324
 - from QR factorization 1697, 2320
 - from RQ factorization 1701, 2328
 - from RZ factorization 1703
 - multiplying by unitary matrix
 - from LQ factorization 1699, 2324
 - from QR factorization 1697, 2320
 - from RQ factorization 1701, 2328
- from RZ factorization 1703
 - QL factorization
 - LAPACK 833
 - ScalAPACK 1971
 - QR factorization
 - with pivoting 806, 809, 1946
 - rank-1 update 87
 - rank-1 update, conjugated 89
 - rank-1 update, unconjugated 91
 - reduction to bidiagonal form 1386, 1407, 2181
 - reduction to upper Hessenberg form 2184
 - RQ factorization
 - LAPACK 845
 - ScalAPACK 2009
 - scalar-matrix-matrix product 139, 395
 - Schur factorization reordering 2163, 2351
 - solving systems of linear equations
 - band storage
 - LAPACK 461
 - ScalAPACK 1895
 - general rectangular distributed matrix
 - computing scaling factors 1938
 - equilibration 1938
 - general rectangular matrix
 - 1-norm value
 - LAPACK 1483
 - ScalAPACK 2219
 - block reflector
 - LAPACK 1547
 - ScalAPACK 2257
 - elementary reflector
 - LAPACK 1545, 2261
 - ScalAPACK 2254
 - Frobenius norm
 - LAPACK 1483
 - ScalAPACK 2219
 - infinity- norm
 - LAPACK 1483
 - ScalAPACK 2219
 - largest absolute value of element
 - LAPACK 1483
 - ScalAPACK 2219
 - LQ factorization
 - LAPACK 1390
 - ScalAPACK 2187
 - multiplication
 - LAPACK 1595
 - ScalAPACK 2283
 - QL factorization
 - LAPACK 1391
 - ScalAPACK 2190
 - QR factorization
 - LAPACK 1393, 1394
 - ScalAPACK 2192
 - reduction of first columns
 - LAPACK 1461, 1464
 - ScalAPACK 2213
 - reduction to bidiagonal form 2199
 - row interchanges
 - LAPACK 1643
 - ScalAPACK 2290
 - RQ factorization
 - LAPACK 1400
 - ScalAPACK 1983, 2194
 - scaling 2230
 - general square matrix
 - reduction to upper Hessenberg form 1388
 - trace 2292
 - general triangular matrix
 - LU factorization

band storage 2172
 general tridiagonal matrix
 1-norm value 1484
 Frobenius norm 1484
 infinity- norm 1484
 largest absolute value of element 1484
 general tridiagonal triangular matrix
 LU factorization
 band storage 2175
 generalized eigenvalue problems
 complex Hermitian positive-definite problem
 band storage 986
 packed storage 982
 complex Hermitian-definite problem 1717, 2348
 real symmetric-definite problem
 band storage 984
 packed storage 980
 See also LAPACK routines, generalized eigenvalue problems 975
 Generalized LLS Problems 1118
 Generalized Nonsymmetric Eigenproblems 1331
 generalized Schur factorization 1457, 1544, 1557, 1558
 Generalized Singular Value Decomposition 1079
 generalized Sylvester equation 1070
 Generalized SymmetricDefinite Eigenproblems 1265
 generation methods 2727
 Geometric 2809
 geqpf 806
 GetBrngProperties 2823
 GetNumRegBrngs 2763
 GetStreamSize 2756
 GetStreamStateBrng 2762
 GFSR 2728
 Givens rotation
 modified Givens transformation parameters 72
 of sparse vectors 177
 parameters 68
 global array 3003
 global index of maximum element of a distributed vector 3006
 global matrix 1873
 gtrfs 557
 Gumbel 2793

H

Helmholtz problem
 three-dimensional 3115
 two-dimensional 3111
 Helmholtz problem on a sphere
 non-periodic 3113
 periodic 3113
 Hermitian band matrix
 1-norm value 1488
 Frobenius norm 1488
 infinity- norm 1488
 largest absolute value of element 1488
 Hermitian distributed matrix
 rank-n update 3071
 scalar-matrix-matrix product 3066
 Hermitian indefinite matrix
 matrix-vector product 1800
 Hermitian matrix
 Bunch-Kaufman factorization
 packed storage 457
 eigenvalues and eigenvectors 2116, 2120, 2123, 2128, 2169
 estimating the condition number
 packed storage 528

generalized eigenvalue problems 975
 inverting the matrix
 packed storage 636
 matrix-vector product
 band storage 93
 packed storage 102
 rank-1 update
 packed storage 104
 rank-2 update
 packed storage 106
 rank-2k update 148
 rank-k update 146
 reducing to standard form
 LAPACK 1715
 ScalAPACK 2345
 reducing to tridiagonal form
 LAPACK 1661, 1717
 ScalAPACK 2293, 2348
 scalar-matrix-matrix product 143
 scaling 2249
 solving systems of linear equations
 packed storage 492
 Hermitian positive definite distributed matrix
 computing scaling factors 1940
 equilibration 1940
 Hermitian positive semidefinite matrix
 Cholesky factorization 432
 Hermitian positive-definite band matrix
 Cholesky factorization 1706
 Hermitian positive-definite distributed matrix
 inverting the matrix 1935
 Hermitian positive-definite matrix
 Cholesky factorization
 band storage 438, 1888
 packed storage 436
 estimating the condition number
 band storage 513
 packed storage 511
 inverting the matrix
 packed storage 617
 solving systems of linear equations
 band storage 474, 1905
 packed storage 472

Hermitian positive-definite tridiagonal matrix
 solving systems of linear equations 1907

Hessenberg matrix
 eigenvalues 2233
 eigenvalues and eigenvectors 2244
 eigenvalues and Schur factorization 2047

heswapr 1712
 hetri2 628
 hetri2x 632
 hgeqz 1050
 Householder matrix
 LAPACK 1550, 1551
 ScalAPACK 2264
 Householder reflector 2369
 hseqr 1011
 Hypergeometric 2813

I

i?amax 77
 i?amin 78
 i?max1 1383
 IBM ESSL library 2827
 IEEE arithmetic 2216
 IEEE standard
 implementation 2420

signbit position 2422
 ila?lr 1738
 iladiag 1860
 ilaenv 1849
 ilaprec 1861
 ilatrans 1861
 ilauplo 1862
 ilaver 1848
 ILU0 preconditioner 2527
 Incomplete LU Factorization Technique 2527
 increment 3365
 iNewAbstractStream 2742
 infinity-norm
 complex Hermitian matrix
 packed storage 1491
 complex Hermitian matrix in RFP format 1752
 complex Hermitian tridiagonal matrix 1492
 complex symmetric matrix 1493
 general rectangular matrix 1483, 2219
 general tridiagonal matrix 1484
 Hermitian band matrix 1488
 real symmetric matrix 1493, 2223
 real symmetric matrix in RFP format 1750
 real symmetric tridiagonal matrix 1492
 symmetric band matrix 1486
 symmetric matrix
 packed storage 1489
 trapezoidal matrix 1500
 triangular band matrix 1496
 triangular matrix
 packed storage 1498
 upper Hessenberg matrix 1485, 2221
 Intel(R) Many Integrated Core Architecture support functions
 mkl_mic_disable 3225
 mkl_mic_enable 3224
 mkl_mic_free_memory 3231
 mkl_mic_get_device_count 3225
 mkl_mic_get_resource_limit 3238
 mkl_mic_get_workdivision 3228
 mkl_mic_register_memory 3232
 mkl_mic_set_device_num_threads 3233
 mkl_mic_set_max_memory 3229
 mkl_mic_set_offload_report 3239
 mkl_mic_set_resource_limit 3235
 mkl_mic_set_workdivision 3226
 mkl_set_env_mode 3249
 Interface Consideration 183
 inverse matrix. inverting a matrix 612, 1932, 1935, 1936
 inverting a matrix
 general matrix
 LAPACK 612
 ScalAPACK 1932
 Hermitian matrix
 packed storage 636
 Hermitian positive-definite matrix
 LAPACK 614
 packed storage 617
 ScalAPACK 1935
 symmetric matrix
 packed storage 634
 symmetric positive-definite matrix
 LAPACK 614
 packed storage 617
 ScalAPACK 1935
 triangular distributed matrix 1936
 triangular matrix
 packed storage 641
 iparmq 1851
 Iterative Sparse Solvers 2496

Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS) 2496

J

Jacobi plane rotations 1245
 Jacobian matrix calculation routines
 ?jacobi 3175
 ?jacobi_delete 3175
 ?jacobi_init 3172
 ?jacobi_solve 3173
 ?jacobix 3177

L

la_gbav 1773
 la_gbrcond 1775
 la_gbrcond_c 1777
 la_gbrcond_x 1779
 la_gercond 1790
 la_gercond_c 1792
 la_gercond_x 1793
 la_hercond_c 1802
 la_hercond_x 1804
 la_lin_berr 1812
 la_porcond 1813
 la_porcond_c 1815
 la_porcond_x 1816
 la_syrcond 1834
 la_syrcond_c 1836
 la_syrcond_x 1837
 LAPACK
 naming conventions 411
 LAPACK auxiliary routines
 ?la_geamv 1788
 ?la_heamv 1800
 ?la_syamv 1832
 ?larscl2 1830
 ?lascl2 1831
 LAPACK routines
 ?gsvj0 1739
 ?gsvj1 1742
 ?hfrk 1746
 ?larfp 1736
 ?sfrk 1744
 2-by-2 generalized eigenvalue problem 1447
 2-by-2 Hermitian matrix
 plane rotation 1544
 2-by-2 orthogonal matrices 1449
 2-by-2 real matrix
 generalized Schur factorization 1457
 2-by-2 real nonsymmetric matrix
 Schur factorization 1502
 2-by-2 symmetric matrix
 plane rotation 1544
 2-by-2 triangular matrix
 singular values 1595
 SVD 1642
 approximation to smallest eigenvalue 1633
 auxiliary routines
 ?gbtf2 1385
 ?gebd2 1386
 ?gehd2 1388
 ?gelq2 1390
 ?geql2 1391
 ?geqr2 1393
 ?geqr2p 1394
 ?gerq2 1400
 ?gesc2 1401

?getc2 1402	?lansy 1493
?getf2 1404	?lantb 1496
?gtts2 1405	?lantp 1498
?hetf2 1722	?lantr 1500
?hetf2_rook 1724	?lanv2 1502
?hfrk 1746	?lapll 1503
?isnan 1406	?lapmr 1504
?la_gbrpvgrw 1787	?lapmt 1505
?la_gerpvgrw 1829	?lapy2 1506
?la_herpvgrw 1811	?lapy3 1507
?la_porpvgrw 1824	?laqgb 1508
?la_syrpvgrw 1845	?laqge 1509
?la_wwaddw 1847	?laqhb 1511
?labrd 1407	?laqhe 1825
?lacgv 1371	?laqhp 1827
?lacn2 1410	?laqp2 1513
?lacon 1412	?laqps 1514
?iacp2 1772	?laqr0 1516
?lacpy 1413	?laqr1 1519
?lacrm 1371	?laqr2 1520
?lacrt 1372	?laqr3 1524
?ladiv 1414	?laqr4 1527
?iae2 1415	?laqr5 1531
?laebz 1416	?laqsb 1534
?laed0 1420	?laqsp 1535
?laed1 1423	?laqsy 1537
?laed2 1424	?laqtr 1538
?laed3 1426	?lar1v 1541
?laed4 1428	?lar2v 1544
?laed5 1430	?larcm 1828
?laed6 1431	?larf 1545
?laed7 1432	?larfb 1547
?laed8 1435	?larfg 1550
?laed9 1438	?larfgp 1551
?laeda 1440	?larfp 1736
?laein 1442	?larft 1553
?laesy 1373	?larfx 1555
?laev2 1444	?largv 1557
?laexc 1446	?larnv 1558
?lag2 1447	?larra 1560
?lags2 1449	?larrb 1561
?lagtf 1452	?larrc 1563
?lagtm 1454	?larrd 1564
?lagts 1455	?larrf 1568
?lagv2 1457	?larrf 1571
?lahef 1651	?larrj 1573
?lahef_rook 1653	?larrk 1575
?lahqr 1459	?larrr 1577
?lahr2 1464	?larrv 1578
?lahrd 1461	?lartg 1581
?laic1 1466	?lartgp 1583
?laisnan 1407	?lartgs 1584
?laln2 1468	?lartv 1586
?lals0 1470	?laruv 1587
?lalsa 1474	?larz 1588
?lalsd 1477	?larzb 1590
?lamrg 1479	?larzt 1592
?laneg 1480	?las2 1595
?langb 1481	?lascl 1595
?lange 1483	?lasd0 1597
?langt 1484	?lasd1 1598
?lanhb 1488	?lasd2 1601
?lanhe 1495	?lasd3 1604
?lanhf 1752	?lasd4 1607
?lanhp 1491	?lasd5 1608
?lanhs 1485	?lasd6 1609
?lansb 1486	?lasd7 1613
?lansf 1750	?lasd8 1617
?lansp 1489	?lasd9 1619
?lanst/?lanht 1492	?lasda 1621

?lasdq 1624
 ?lasdt 1627
 ?laset 1627
 ?lasq1 1629
 ?lasq2 1630
 ?lasq3 1631
 ?lasq4 1633
 ?lasq5 1634
 ?lasq6 1636
 ?lasr 1637
 ?lasrt 1640
 ?lassq 1641
 ?lasv2 1642
 ?laswp 1643
 ?lasy2 1644
 ?lasyf 1647
 ?lasyf_rook 1649
 ?latbs 1655
 ?latdf 1657
 ?latps 1659
 ?latrd 1661
 ?latrs 1663
 ?latrz 1667
 ?lauu2 1669
 ?lauum 1670
 ?orbdb/?unbdb
 1098
 ?orbdb1/?unbdb1
 1671
 ?orbdb2/?unbdb2
 1674
 ?orbdb3/?unbdb3
 1678
 ?orbdb4/?unbdb4
 1681
 ?orbdb5/?unbdb5
 1684
 ?orbdb6/?unbdb6
 1687
 ?orcsd/?uncsd
 1255
 ?orcsd2by1/?
 uncsd2by
 1 1261
 ?org2l/?ung2l
 1689
 ?org2r/?ung2r
 1691
 ?orgl2l/?ungl2
 1692
 ?orgr2/?ungr2
 1694
 ?orm2l/?unm2l
 1695
 ?orm2r/?unm2r
 1697
 ?orml2/?unml2
 1699
 ?ormr2/?unmr2
 1701
 ?ormr3/?unmr3
 1703
 ?pbtf2 1706
 ?potf2 1707
 ?pstf2 1768
 ?ptts2 1708
 ?rot 1375
 ?rscl 1710
 ?sfrk 1744
 ?spmv 1376
 ?spr 1377
 ?sum1 1384
 ?sygs2/?hegs2
 1715
 ?symv 1380
 ?syr 1382
 ?sytd2/?hetd2
 1717
 ?sytf2 1719
 ?sytf2_rook 1721
 ?tftp 1753
 ?tfttr 1754
 ?tge2 1725
 ?tgzy2 1727
 ?tptrf 1762
 ?tpptr 1764
 ?trti2 1731
 ?trttf 1765
 ?trttp 1767
 clag2z 1732
 dlag2s 1733
 dlat2s 1770
 i?max1 1383
 ila?lc 1737
 ila?lr 1738
 slag2d 1734
 zlag2c 1735
 zlat2c 1771
 banded matrix equilibration
 ?gbequ 647
 ?gbequb 650
 bidiagonal divide and conquer 1627
 block reflector
 triangular factor 1553, 1592
 checking for safe infinity 1853
 complex Hermitian matrix
 packed storage 1491
 complex Hermitian matrix in RFP format
 1752
 complex Hermitian tridiagonal matrix 1492
 complex matrix multiplication 1371, 1828
 complex symmetric matrix
 computing eigenvalues and
 eigenvectors 1373
 matrix-vector product 1380
 symmetric rank-1 update 1382
 complex symmetric packed matrix
 symmetric rank-1 update 1377
 complex vector
 1-norm using true absolute value
 1384
 index of element with max absolute
 value 1383
 linear transformation 1372
 matrix-vector product 1376
 plane rotation 1375
 complex vector conjugation 1371
 condition number estimation
 ?disna 973
 ?gbcon 504
 ?gecon 502
 ?gtcon 506
 ?hecon 522
 ?hecon_rook 524
 ?hpcon 528
 ?pbcon 513
 ?pocon 509
 ?ppcon 511
 ?ptcon 516
 ?spcon 526

?sycon 518
?sycon_rook 520
?tbcn 534
?tpcon 532
?trcon 530
determining machine parameters 1856
diagonally dominant triangular factorization
?dttrfb 428
dqd transform 1636
dqds transform 1634
driver routines
generalized LLS problems
?ggglm 1121
?gglse 1118
generalized nonsymmetric eigenproblems
?gges 1332
?ggesx 1338
?ggev 1345
?ggevx 1350
generalized symmetric definite eigenproblems
?hbgy 1312
?hbgyd 1319
?hbgyx 1327
?hegv 1268
?hegvd 1275
?hegvx 1284
?hpvgv 1291
?hpgvd 1297
?hpgvx 1306
?sbgy 1310
?sbgyd 1315
?sbgyx 1323
?spgv 1289
?spgvd 1294
?spgvy 1301
?sygv 1265
?sygvd 1271
?sygvx 1279
linear least squares problems
?gels 1104
?gelsd 1114
?gelss 1111
?gelsy 1107
?lals0 (auxiliary) 1470
?lalsa (auxiliary) 1474
?lalsd (auxiliary) 1477
nonsymmetric eigenproblems
?gees 1208
?geesx 1213
?geev 1219
?geevx 1223
singular value decomposition
?gejsv 1238
?gelsd 1114
?gesdd 1234
?gesvd 1229
?gesvj 1245
?ggsvd 1249
solving linear equations
?dtsvb 707
?gbsv 684
?gbsvx 686
?gbsvxx 692
?gesv 665
?gesvx 669
?gesvxx 675
?gtsv 701
?gtsvx 702
?hesv 764
?hesv_rook 767
?hesvx 769
?hesvxx 774
?hpsv 788
?hpsvx 790
?pbsv 733
?pbsvx 735
?posv 709
?posvx 713
?posvxx 718
?ppsv 726
?ppsvx 728
?ptsv 740
?ptsvx 742
?spsv 781
?spsvx 783
?sysv 746
?sysv_rook 749
?sysvx 751
?sysvxx 756
symmetric eigenproblems
?hbev 1176
?hbevd 1182
?hbevx 1190
?heev 1127
?heevd 1133
?heevr 1150
?heevx 1141
?hpev 1157
?hpevd 1163
?hpevx 1170
?sbev 1174
?sbevd 1179
?sbevx 1186
?spev 1155
?spevd 1159
?spevx 1166
?stev 1194
?stevd 1196
?stevr 1203
?stevx 1200
?syev 1125
?syevd 1130
?syevr 1145
?syevx 1137
environmental enquiry 1849, 1851
finding a relatively isolated eigenvalue 1571
general band matrix equilibration 1508
general matrix
?lagge 1864
?lagsy 1866
?latms 1867
block reflector 1590
elementary reflector 1588
reduction to bidiagonal form 1386, 1407
general matrix equilibration
?geeque 643
?geequeb 645
general rectangular matrix
block reflector 1547
elementary reflector 1545, 1555
equilibration 1509, 1825, 1827

LQ factorization 1390
 plane rotation 1637
 QL factorization 1391
 QR factorization 1393, 1394
 row interchanges 1643
 RQ factorization 1400
 general square matrix
 reduction to upper Hessenberg form
 1388
 general tridiagonal matrix 1452, 1454, 1455,
 1484, 1568, 1578
 generalized eigenvalue problems
 ?hbgst 986
 ?hegst 977
 ?hpdst 982
 ?pbstf 989
 ?sbgst 984
 ?spgst 980
 ?sygst 975
 generalized SVD
 ?ggsvp 1080
 ?tgsja 1084
 generalized Sylvester equation
 ?tgsyl 1070
 Hermitian band matrix
 equilibration 1511, 1537
 Hermitian band matrix in packed storage
 equilibration 1535
 Hermitian indefinite matrix equilibration
 ?heequb 663
 Hermitian matrix
 ?laghe 1865
 computing eigenvalues and
 eigenvectors 1444
 Hermitian positive-definite matrix
 equilibration
 ?poequ 652
 ?poequb 654
 Householder matrix
 elementary reflector 1550, 1551
 ila?lc 1737
 ila?lr 1738
 incremental condition estimation 1466
 linear dependence of vectors 1503
 LQ factorization
 ?gelq2 1390
 ?gelqf 821
 ?orglq 824
 ?ormlq 826
 ?unglq 829
 ?unmlq 831
 LU factorization
 general band matrix 1385
 matrix equilibration
 ?laqgb 1508
 ?laqge 1509
 ?laqhb 1511
 ?laqhe 1825
 ?laqhp 1827
 ?laqsb 1534
 ?laqsp 1535
 ?laqsy 1537
 ?pbequ 658
 ?ppequ 656
 matrix inversion
 ?getri 612
 ?hetri 622
 ?hetri_rook 624
 ?hetri2 628
 ?hetri2x 632
 ?hptri 636
 ?potri 614
 ?pptri 617
 ?sptri 634
 ?systri 619
 ?systri_rook 620
 ?systri2 626
 ?systri2x 630
 ?tptri 641
 ?trtri 638
 matrix-matrix product
 ?lagtm 1454
 merging sets of singular values 1601, 1613
 mixed precision iterative refinement
 subroutines 665, 709, 1732–1735
 nonsymmetric eigenvalue problems
 ?gebak 1009
 ?gebal 1006
 ?gehrd 993
 ?hsein 1016
 ?hseqr 1011
 ?orghr 996
 ?ormhr 998
 ?trevc 1021
 ?trexc 1031
 ?trsen 1033
 ?trsna 1026
 ?unghr 1001
 ?unmhr 1003
 off-diagonal and diagonal elements 1627
 permutation list creation 1479
 permutation of matrix columns 1505
 permutation of matrix rows 1504
 plane rotation 1581, 1583, 1584, 1586, 1637
 plane rotation vector 1557
 QL factorization
 ?geql2 1391
 ?geqlf 833
 ?orgql 836
 ?ormql 840
 ?ungql 838
 ?unmql 842
 QR factorization
 ?gemqrt 804
 ?geqp3 809
 ?geqpf 806
 ?geqr2 1393
 ?geqr2p 1394
 ?geqrf 796
 ?geqrfp 799
 ?geqrt 802
 ?geqrt2 1396
 ?geqrt3 1398
 ?ggqrf 865
 ?ggrqf 868
 ?laqp2 1513
 ?laqps 1514
 ?orgqr 812
 ?ormqr 814
 ?tpmqrt 874
 ?tpqrt 872
 ?tpqrt2 1756
 ?tprfb 1759
 ?ungqr 816
 ?unmqr 819
 ?pgeqrf 1943
 random numbers vector 1558
 real lower bidiagonal matrix
 SVD 1624
 real square bidiagonal matrix

singular values 1629
 real symmetric matrix 1493
 real symmetric matrix in RFP format 1750
 real symmetric tridiagonal matrix 1416, 1492
 real upper bidiagonal matrix
 singular values 1597
 SVD 1598, 1621, 1624
 real upper quasi-triangular matrix
 orthogonal similarity transformation
 1446
 reciprocal condition numbers for
 eigenvalues and/or
 eigenvectors
 ?tgsna 1075
 rectangular full packed format 434, 470
 RQ factorization
 ?geqr2 1400
 ?gerqf 845
 ?orgqr 848
 ?ormrq 851
 ?ungqr 849
 ?unmrq 854
 RZ factorization
 ?ormrz 859
 ?tzrzf 856
 ?unmrz 862
 singular value decomposition
 ?bdscd 902
 ?bdsgqr 898
 ?gbbrd 883
 ?gebrd 879
 ?orgbr 886
 ?ormbr 889
 ?ungbr 892
 ?unmbr 895
 solution refinement and error estimation
 ?gbrfs 547
 ?gbrfsx 550
 ?gerfs 537
 ?gerfsx 540
 ?gttrfs 557
 ?herfs 588
 ?herfsx 591
 ?hprrfs 600
 ?la_gbrfsx_extended 1781
 ?la_gerfsx_extended 1794
 ?la_herfsx_extended 1805
 ?la_porfsx_extended 1818
 ?la_syrfsx_extended 1839
 ?pbrfs 573
 ?porfs 560
 ?porfsx 563
 ?pprfs 570
 ?pttrfs 576
 ?sprfs 597
 ?syrfs 579
 ?syrfsx 582
 ?tbrfs 608
 ?tprfs 605
 ?trrfs 603
 solving linear equations
 ?dttrsrb 466
 ?gbtrs 461
 ?getrs 459
 ?gttrs 464
 ?heswapr 1712
 ?hetrs 482
 ?hetrs_rook 484
 ?hetrs2 488
 ?hptrs 492
 ?iln2 1468
 ?laqtr 1538
 ?pbtrs 474
 ?pftrs 470
 ?potrs 468
 ?pptrs 472
 ?ptrs 476
 ?sptrs 490
 ?syswapr 1711
 ?syswapr1 1714
 ?sytrs 478
 ?sytrs_rook 480
 ?sytrs2 486
 ?tbtrs 499
 ?tptrs 497
 ?trtrs 494
 sorting numbers 1640
 square root 1506, 1507
 square roots 1604, 1607, 1608, 1617, 1619,
 1854
 Sylvester equation
 ?lasy2 1644
 ?tgsy2 1727
 ?trsy2 1038
 symmetric band matrix
 equilibration 1534, 1537
 symmetric band matrix in packed
 storage
 equilibration 1535
 symmetric eigenvalue problems
 ?disna 973
 ?hbtrd 942
 ?herdb 914
 ?hetrd 921
 ?hptrd 934
 ?opgtr 930
 ?opmtr 932
 ?orgtr 916
 ?ormtr 918
 ?pteqr 964
 ?sbtrd 940
 ?sptrd 928
 ?stebz 967
 ?stedc 954
 ?steqr 959
 ?stein 970
 ?stemr 950
 ?steqr 946
 ?sterf 945
 ?syrd2 912
 ?sytrd 909
 ?ungtr 924
 ?unmtr 926
 ?upgtr 936
 ?upmtr 938
 auxiliary
 ?iae2 1415
 ?laebz 1416
 ?laedo 1420
 ?laed1 1423
 ?laed2 1424
 ?laed3 1426
 ?laed4 1428
 ?laed5 1430
 ?laed6 1431
 ?laed7 1432
 ?laed8 1435
 ?laed9 1438
 ?laeda 1440
 symmetric indefinite matrix equilibration

?syequb 661
symmetric matrix
 computing eigenvalues and eigenvectors 1444
 packed storage 1489
symmetric positive-definite matrix
 equilibration
 ?poequ 652
 ?poequb 654
symmetric positive-definite tridiagonal matrix
 eigenvalues 1630
test routines
 ?lagge 1864
 ?laghe 1865
 ?lagsy 1866
 ?latms 1867
trapezoidal matrix 1500, 1667
 triangular factorization
 ?gbtrf 424
 ?getrf 421
 ?gttrf 426
 ?hetrf 449
 ?hetrf_rook 451
 ?hptrf 457
 ?pbtrf 438
 ?potrf 430
 ?pptrf 436
 ?pstrf 432
 ?pttrf 440
 ?sptrf 454
 ?sytrf 441
 ?sytrf_rook 445
 p?dbtrf 1882
 triangular matrix
 packed storage 1498
 triangular matrix factorization
 ?pftrf 434
 ?pftri 615
 ?tftri 639
triangular system of equations 1659, 1663
tridiagonal band matrix 1496
uniform distribution 1587
unreduced symmetric tridiagonal matrix
 1420
 updated upper bidiagonal matrix
 SVD 1609
updating sum of squares 1641
 upper Hessenberg matrix
 computing a specified eigenvector 1442
 eigenvalues 1459
 Schur factorization 1459
utility functions and routines
 ?labad 1854
 ?lamc1 1856
 ?lamc2 1856
 ?lamc3 1857
 ?lamc4 1858
 ?lamc5 1859
 ?lamch 1855
 chla_transtype 1859
 ieeck 1853
 iladiag 1860
 ilaenv 1849
 ilaprec 1861
 ilatrans 1861
 ilauplo 1862
 ilaver 1848
 iparmq 1851
 xerbla_array 1863
Laplace 2779
Laplace problem
 three-dimensional 3116
 two-dimensional 3112
largest absolute value of element
 complex Hermitian matrix
 packed storage 1491
 complex Hermitian matrix in RFP format 1752
 complex Hermitian tridiagonal matrix 1492
 complex symmetric matrix 1493
 general rectangular matrix 1483, 2219
 general tridiagonal matrix 1484
 Hermitian band matrix 1488
 real symmetric matrix 1493, 2223
 real symmetric matrix in RFP format 1750
 real symmetric tridiagonal matrix 1492
 symmetric band matrix 1486
 symmetric matrix
 packed storage 1489
 trapezoidal matrix 1500
 triangular band matrix 1496
 triangular matrix
 packed storage 1498
 upper Hessenberg matrix 1485, 2221
leading dimension 3368
leapfrog method 2732
LeapfrogStream 2757
least squares problems 793
length. dimension 3365
library version 3184
Library Version Obtaining 3184
library version string 3186
linear combination of distributed vectors 3009
linear combination of vectors 57, 388
Linear Congruential Generator 2728
linear equations, solving
 tridiagonal symmetric positive-definite matrix
 LAPACK 740
 ScalAPACK 2094
 band matrix
 LAPACK 684, 686
 ScalAPACK 2075
 banded matrix
 extra precise iterative refinement
 LAPACK 692
 extra precise iterative refinement 550, 1781, 1818
 LAPACK 692
 Cholesky-factored matrix
 LAPACK 474
 ScalAPACK 1905
 diagonally dominant tridiagonal matrix
 LAPACK 466, 707
 diagonally dominant-like matrix
 banded 1898
 tridiagonal 1900
 general band matrix
 ScalAPACK 2078
 general matrix
 band storage 461, 1895
 extra precise iterative refinement 540
 extra precise iterative refinement 1794
 general tridiagonal matrix
 ScalAPACK 2080
 Hermitian indefinite matrix
 extra precise iterative refinement
 LAPACK 774
 extra precise iterative refinement 1805
 LAPACK 774
 Hermitian matrix

error bounds 769, 790
 packed storage 492, 788, 790
 Hermitian positive-definite matrix
 band storage
 LAPACK 733
 ScaLAPACK 2091
 error bounds
 LAPACK 713
 ScaLAPACK 2085
 extra precise iterative refinement
 LAPACK 718
 LAPACK
 linear equations, solving
 packed storage 472, 726, 728
 ScaLAPACK 2085
 Hermitian positive-definite tridiagonal linear equations 241
 Hermitian positive-definite tridiagonal matrix 1907
 multiple right-hand sides
 band matrix
 LAPACK 684, 686
 ScaLAPACK 2075
 banded matrix
 LAPACK 692
 diagonally dominant tridiagonal matrix 707
 Hermitian indefinite matrix
 LAPACK 774
 Hermitian matrix 764, 767, 788
 Hermitian positive-definite matrix
 band storage 733
 square matrix
 LAPACK 665, 669, 675
 ScaLAPACK 2067, 2069
 symmetric indefinite matrix
 LAPACK 756
 symmetric matrix 746, 749, 781
 symmetric positive-definite matrix
 band storage 733
 symmetric/Hermitian positive-definite matrix
 LAPACK 718
 tridiagonal matrix 701, 702
 overestimated or underestimated system 2097
 square matrix
 error bounds
 LAPACK 669, 686
 ScaLAPACK 2069
 extra precise iterative refinement
 LAPACK 675
 LAPACK 665, 669, 675
 ScaLAPACK 2067, 2069
 symmetric indefinite matrix
 extra precise iterative refinement
 LAPACK 756
 extra precise iterative refinement 1839
 LAPACK 756
 symmetric matrix
 error bounds 751, 783
 packed storage 490, 781, 783
 symmetric positive-definite matrix
 band storage
 LAPACK 733
 ScaLAPACK 2091
 error bounds
 LAPACK 713

ScaLAPACK 2085
 extra precise iterative refinement
 LAPACK 563, 718
 LAPACK 709, 713, 718
 packed storage 472, 726, 728
 ScaLAPACK 2083, 2085
 symmetric positive-definite tridiagonal linear equations 2415
 triangular matrix
 band storage 499, 2332
 packed storage 497
 tridiagonal Hermitian positive-definite matrix
 error bounds 742
 LAPACK 740
 multiple right-hand sides
 ScaLAPACK 2094
 std::vector
 symmetric error bounds 702
 LAPACK 464, 476, 701, 702
 LAPACK auxiliary 1541
 ScaLAPACK auxiliary 2413
 tridiagonal symmetric positive-definite matrix
 error bounds 742
 Linear Least Squares (LLS) Problems 1104
 LoadStreamF 2753
 LoadStreamM 2755
 Lognormal 2790
 LQ factorization
 computing the elements of
 orthogonal matrix Q 824
 real orthogonal matrix Q 1961
 unitary matrix Q 829, 1963
 general rectangular matrix 1390, 2187
 Isame 3202
 Isamen 3202
 LU factorization
 band matrix
 blocked algorithm 2410
 unblocked algorithm 2408
 diagonally dominant tridiagonal matrix 428
 diagonally dominant-like tridiagonal matrix 1884
 general band matrix 1385
 general matrix 1404, 2197
 solving linear equations
 general matrix 1401
 square matrix 2069
 tridiagonal matrix 1405, 1455
 triangular band matrix 2172
 tridiagonal band matrix 2175
 tridiagonal matrix 426, 1452, 2412
 with complete pivoting 1402, 1657
 with partial pivoting 1404, 2197

M

machine parameters
 LAPACK 1855
 ScaLAPACK 2421
 matrices, copying
 general matrix 2423
 trapezoidal matrix 2425
 matrix arguments
 column-major ordering 3365, 3368
 example 3369
 leading dimension 3368
 number of columns 3368
 number of rows 3368
 transposition parameter 3368
 matrix block
 QR factorization
 with pivoting 1513

matrix converters
 mkl_?csrbsr 360
 mkl_?csrcoo 357
 mkl_?csrcsc 364
 mkl_?csrdia 367
 mkl_?csrsky 371
 mkl_?dnsCSR 353
 matrix equation
 AX = B 164, 419, 459, 1748, 1875, 1893
 matrix one-dimensional substructures 3365
 matrix-matrix operation
 product
 general distributed matrix 3063
 general matrix 139, 395
 rank-2k update
 Hermitian distributed matrix 3071
 Hermitian matrix 148
 symmetric distributed matrix 3079
 symmetric matrix 158
 rank-k update
 Hermitian matrix 146
 symmetric distributed matrix 3076
 rank-n update
 symmetric matrix 155
 scalar-matrix-matrix product
 Hermitian distributed matrix 3066
 Hermitian matrix 143
 symmetric distributed matrix 3073
 symmetric matrix 151
 matrix-matrix operation:scalar-matrix-matrix product
 triangular distributed matrix 3086
 triangular matrix 161
 matrix-vector operation
 product
 Hermitian matrix 93, 96, 102
 real symmetric matrix 111, 117
 triangular matrix 123, 129, 134
 rank-1 update
 Hermitian matrix 98, 104
 real symmetric matrix 113, 119
 rank-2 update
 Hermitian matrix 100, 106
 symmetric matrix 115, 121
 matrix-vector operation:product
 Hermitian matrix
 band storage 93
 packed storage 102
 real symmetric matrix
 packed storage 111
 symmetric matrix
 band storage 108
 triangular matrix
 band storage 123
 packed storage 129
 matrix-vector operation:rank-1 update
 Hermitian matrix
 packed storage 104
 real symmetric matrix
 packed storage 113
 matrix-vector operation:rank-2 update
 Hermitian matrix
 packed storage 106
 symmetric matrix
 packed storage 115
 mkl_?bsrgemv 194
 mkl_?bsrmm 292
 mkl_?bsrmv 259
 mkl_?bsrsm 318
 mkl_?bsrsv 276
 mkl_?bsrsymv 206
 mkl_?bsrtrsv 218
 mkl_?coogemv 197
 mkl_?coomm 302
 mkl_?coomv 268
 mkl_?coosm 315
 mkl_?coosv 284
 mkl_?coosymv 209
 mkl_?cootrv 221
 mkl_?cscmm 297
 mkl_?cscmv 264
 mkl_?cscsm 310
 mkl_?cscsv 280
 mkl_?csradd 375
 mkl_?csrbsr 360
 mkl_?csrcoo 357
 mkl_?csrcsc 364
 mkl_?csrdia 367
 mkl_?csrgemv 191
 mkl_?csrmm 288
 mkl_?csrmultcsr 379
 mkl_?csrmultd 384
 mkl_?csrmv 255
 mkl_?csrsky 371
 mkl_?csrsm 306
 mkl_?csrsv 272
 mkl_?csrsymv 203
 mkl_?csrtrsv 215
 mkl_?diagemv 200
 mkl_?diamm 337
 mkl_?diamr 322
 mkl_?diasm 346
 mkl_?diasv 330
 mkl_?diasymv 212
 mkl_?diatrv 224
 mkl_?dnsCSR 353
 mkl_?imatcopy 398
 mkl_?omatadd 407
 mkl_?omatcopy 401
 mkl_?omatcopy2 404
 mkl_?skymm 341
 mkl_?skymv 326
 mkl_?skysm 350
 mkl_?skysv 334
 mkl_calloc 3213
 MKL_calloc 3213
 mkl_cbwr_get 3241
 mkl_cbwr_get_auto_branch 3242
 mkl_cbwr_set 3240
 mkl_cspblas_?bsrgemv 231
 mkl_cspblas_?bsrsymv 240
 mkl_cspblas_?bsrtrsv 249
 mkl_cspblas_?coogemv 234
 mkl_cspblas_?coosymv 243
 mkl_cspblas_?csrgemv 228
 mkl_cspblas_?csrsymv 237
 mkl_cspblas_?csrtrsv 246
 mkl_cspblas_?dcootrv 252
 mkl_disable_fast_mm 3209
 MKL_Disable_Fast_MM 3209
 mkl_domain_get_max_threads 3195
 MKL_Domain_Get_Max_Threads 3195
 mkl_domain_set_num_threads 3189
 mkl_enable_instructions 3248
 MKL_Enable_Instructions 3248
 mkl_free
 usage example 3216
 MKL_free 3215
 mkl_free_buffers 3207
 MKL_Free_Buffers 3207
 mkl_get_clocks_frequency 3206

MKL_Get_Clocks_Frequency 3206
 mkl_get_cpu_clocks 3204
 MKL_Get_Cpu_Clocks 3204
 mkl_get_cpu_frequency 3205
 MKL_Get_Cpu_Frequency 3205
 mkl_get_dynamic 3196
 MKL_Get_Dynamic 3196
 mkl_get_max_cpu_frequency 3206
 MKL_Get_Max_Cpu_Frequency 3206
 mkl_get_max_threads 3194
 MKL_Get_Max_Threads 3194
 mkl_get_version 3184
 MKL_Get_Version 3184
 mkl_get_version_string 3186
 mkl_malloc
 usage example 3216
 MKL_malloc 3212
 mkl_mem_stat
 usage example 3216
 MKL_Mem_Stat 3210
 mkl_mic_disable 3225
 mkl_mic_enable 3224
 mkl_mic_free_memory 3231
 mkl_mic_get_device_count 3225
 mkl_mic_get_resource_limit 3238
 mkl_mic_get_workdivision 3228
 mkl_mic_register_memory 3232
 mkl_mic_set_device_num_threads 3233
 mkl_mic_set_max_memory 3229
 mkl_mic_set_offload_report 3239
 mkl_mic_set_resource_limit 3235
 mkl_mic_set_workdivision 3226
 mkl_pardiso_pivot 2444
 mkl_peak_mem_usage 3211
 mkl_progress 3246
 mkl_realloc 3214
 MKL_realloc 3214
 mkl_set_dynamic 3193
 MKL_Set_Dynamic 3193
 mkl_set_env_mode 3249
 mkl_set_exit_handler 3200
 mkl_set_interface_layer 3219
 mkl_set_num_threads 3188
 MKL_Set_Num_Threads 3188
 mkl_set_num_threads_local 3191
 MKL_Set_Num_Threads_Local 3191
 mkl_set_progress 3222
 mkl_set_threading_layer 3220
 mkl_set_xerbla 3221
 mkl_thread_free_buffers 3208
 MKL_Thread_Free_Buffers 3208
 mkl_verbose 3250
 MKLFreeTIs 2717
 MPI 1873
 Multiplicative Congruential Generator 2728

N

naming conventions
 BLAS 51
 LAPACK 1875
 Nonlinear Optimization Solvers 3154
 PBLAS 3004
 Sparse BLAS Level 1 168
 Sparse BLAS Level 2 181
 Sparse BLAS Level 3 181
 VML 2576
 negative eigenvalues 2216
 NegBinomial 2819
 NewStream 2739

NewStreamEx 2741
 NewTaskX1D 2841
 Nonsymmetric Eigenproblems 1208

O

off-diagonal elements
 initialization 2285
 LAPACK 1627
 ScalAPACK 2285
 orthogonal matrix
 CS decomposition
 LAPACK 1091, 1098, 1255, 1261, 1671,
 1674, 1678, 1681, 1684, 1687
 from LQ factorization
 LAPACK 1692
 ScalAPACK 2312
 from QR factorization
 LAPACK 1691
 ScalAPACK 2309
 from RQ factorization
 LAPACK 1694
 ScalAPACK 2314
 multiplication 2160

P

p?agemm 3025
 p?ahemv 3036
 p?amax 3006
 p?asum 3007
 p?asymv 3044
 p?atrmv 3053
 p?axpy 3009
 p?copy 3010
 p?dbsv 2078
 p?dbtrf 1882
 p?dbtrs 1898
 p?dbtrsv 2172
 p?dot 3012
 p?dotc 3014
 p?dotu 3015
 p?dtsv 2080
 p?dttrf 1884
 p?dttrs 1900
 p?dttrsv 2175
 p?gbsv 2075
 p?gbtrf 1879
 p?gbtrs 1895
 p?geadd 3059
 p?gebd2 2181
 p?gebrd 2050
 p?gecon 1912
 p?geequ 1938
 p?gehd2 2184
 p?gehrd 2035
 p?gelq2 2187
 p?gelqf 1958
 p?gels 2097
 p?gemm 3063
 p?gemr2d 2423
 p?gemv 3022
 p?geql2 2190
 p?geqlf 1971
 p?geqpf 1946
 p?geqr2 2192
 p?geqrf 1943

p?ger 3028
 p?gerc 3030
 p?gerfs 1921
 p?gerq2 2194
 p?gerqf 1983
 p?geru 3032
 p?gesv 2067
 p?gesvd 2135
 p?gesvx 2069
 p?getf2 2197
 p?getrf 1877
 p?getri 1932
 p?getrs 1893
 p?ggqrdf 2005
 p?ggrqf 2009
 p?heev 2116
 p?heevd 2120
 p?heevx 2128
 p?hegst 2065
 p?hegvx 2146
 p?hemm 3066
 p?hemv 3034
 p?her 3038
 p?her2 3040
 p?her2k 3071
 p?herk 3069
 p?hetrd 2021
 p?labad 2419
 p?labrd 2199
 p?lachkieee 2420
 p?lacon 2203
 p?laconsb 2205
 p?lacp2 2206
 p?lacp3 2208
 p?lacpy 2209
 p?laevswp 2211
 p?lahqr 2045
 p?lahrd 2213
 p?laiect 2216
 p?lamch 2421
 p?lange 2219
 p?lanhs 2221
 p?lantr 2225
 p?lapiv 2227
 p?laqge 2230
 p?laqsy 2249
 p?lared1d 2251
 p?lared2d 2253
 p?larf 2254
 p?larfb 2257
 p?larfc 2261
 p?larfg 2264
 p?larft 2266
 p?larz 2269
 p?larzb 2273
 p?larzt 2280
 p?lascl 2283
 p?laset 2285
 p?lasmsub 2287
 p?lasnbt 2422
 p?lassq 2288
 p?laswp 2290
 p?latra 2292
 p?latrd 2293
 p?latrz 2300
 p?lauu2 2302
 p?lauum 2304
 p?lawil 2305
 p?max1 2167
 p?nrm2 3017
 p?org2l/p?ung2l 2307
 p?org2r/p?ung2r 2309
 p?orgl2/p?ungl2 2312
 p?orglq 1961
 p?orgql 1973
 p?orgqr 1948
 p?orgr2/p?ungr2 2314
 p?orgrq 1986
 p?orm2l/p?unm2l 2317
 p?orm2r/p?unm2r 2320
 p?ormbr 2054
 p?ormhr 2038
 p?orml2/p?unml2 2324
 p?ormlq 1965
 p?ormql 1977
 p?ormqr 1952
 p?ormr2/p?unmr2 2328
 p?ormrq 1989
 p?ormrz 1998
 p?ormtr 2017
 p?pbsv 2091
 p?pbtrf 1888
 p?pbtrs 1905
 p?pbtrsv 2332
 p?pocon 1915
 p?poequ 1940
 p?porfs 1925
 p?posv 2083
 p?posvx 2085
 p?potf2 2340
 p?potrf 1886
 p?potri 1935
 p?potrs 1903
 p?ptsv 2094
 p?pttrf 1890
 p?pttrs 1907
 p?pttrsv 2336
 p?rscl 2344
 p?scal 3018
 p?stebz 2027
 p?stein 2031
 p?sum1 2171
 p?swap 3019
 p?syev 2100
 p?syevd 2103
 p?syevx 2110
 p?sygs2/p?hegs2 2345
 p?sygst 2063
 p?sygvx 2139
 p?symm 3073
 p?symv 3042
 p?syr 3047
 p?syr2 3048
 p?syr2k 3079
 p?syrk 3076
 p?sytd2/p?hett2 2348
 p?sytrd 2014
 p?tradd 3061
 p?tran 3082
 p?tranc 3085
 p?tranu 3083
 p?trcon 1918
 p?trmm 3086
 p?trmr2d 2425
 p?trmv 3051
 p?trrfs 1928
 p?trsm 3089
 p?trsv 3056
 p?trti2 2359
 p?trtri 1936

p?trtrs 1909
 p?tzrf 1996
 p?unglq 1963
 p?ungql 1975
 p?ungqr 1950
 p?ungrq 1988
 p?unmbr 2058
 p?unmhr 2042
 p?unmlq 1968
 p?unmql 1980
 p?unmqr 1955
 p?unmrq 1992
 p?unmrz 2002
 p?unmtr 2024
 Packed formats 2946
 packed storage scheme 3366
 parallel direct solver (PARDISO) 2429
 Parallel Direct Sparse Solver for Clusters 2467
 parallel direct sparse solver interface
 mkl_pardiso_pivot 2444
 pardiso 2434
 pardiso_64 2442
 pardiso_getdiag 2445
 pardiso_getenv 2443
 pardiso_handle_delete 2449
 pardiso_handle_delete_64 2452
 pardiso_handle_restore 2448
 pardiso_handle_restore_64 2451
 pardiso_handle_store 2447
 pardiso_handle_store_64 2450
 pardiso_setenv 2443
 pardisoinit 2440
 parallel direct sparse solver interface for clusters
 cluster_sparse_solver 2469
 parameters
 for a Givens rotation 68
 modified Givens transformation 72
 pardiso 2434
 pardiso iparm parameter 2457
 PARDISO parameters 2452
 pardiso_64 2442
 PARDISO_DATA_TYPE 2467
 pardiso_getdiag 2445
 pardiso_getenv 2443
 pardiso_handle_delete 2449
 pardiso_handle_delete_64 2452
 pardiso_handle_restore 2448
 pardiso_handle_restore_64 2451
 pardiso_handle_store 2447
 pardiso_handle_store_64 2450
 pardiso_setenv 2443
 PARDISO* solver 2429
 pardisoinit 2440
 Partial Differential Equations support
 Helmholtz problem on a sphere 3113
 Poisson problem on a sphere 3113
 three-dimensional Helmholtz problem 3115
 three-dimensional Laplace problem 3116
 three-dimensional Poisson problem 3116
 two-dimensional Helmholtz problem 3111
 two-dimensional Laplace problem 3112
 two-dimensional Poisson problem 3112
 PBLAS Level 1 functions
 p?amax 3006
 p?asum 3007
 p?dot 3012
 p?dotc 3014
 p?dotu 3015
 p?nrm2 3017
 PBLAS Level 1 routines
 p?amax 3006
 p?asum 3006
 p?axpy 3006, 3009
 p?copy 3006, 3010
 p?dot 3006
 p?dotc 3006
 p?dotu 3006
 p?nrm2 3006
 p?scal 3006, 3018
 p?swap 3006, 3019
 PBLAS Level 2 routines
 ?agemv 3021
 ?asymv 3021
 ?gemv 3021
 ?ger 3021
 ?gerc 3021
 ?geru 3021
 ?hemv 3021
 ?her 3021
 ?her2 3021
 ?symv 3021
 ?syr 3021
 ?syr2 3021
 ?trmv 3021
 ?trsv 3021
 p?agemv 3025
 p?ahemv 3036
 p?asymv 3044
 p?atrmv 3053
 p?gemv 3022
 p?ger 3028
 p?gerc 3030
 p?geru 3032
 p?hemv 3034
 p?her 3038
 p?her2 3040
 p?symv 3042
 p?syr 3047
 p?syr2 3048
 p?trmv 3051
 p?trsv 3056
 PBLAS Level 3 routines
 p?geadd 3059
 p?gemm 3058, 3063
 p?hemm 3058, 3066
 p?her2k 3058, 3071
 p?herk 3058, 3069
 p?symm 3058, 3073
 p?syr2k 3058, 3079
 p?syrk 3058, 3076
 p?tradd 3061
 p?tran 3082
 p?tranc 3085
 p?tranu 3083
 p?trmm 3058, 3086
 p?trsm 3058, 3089
 PBLAS routines
 routine groups 3003
 pcagemv 3025
 pcahemv 3036
 pcamax 3006
 pcatrmv 3053
 pcaxpy 3009
 pccopy 3010
 pcdotc 3014
 pcdotu 3015
 pcgeadd 3059
 pcgecon 1912
 pcgemm 3063
 pcgemv 3022

pcgerc 3030
 pcgeru 3032
 pcheevr 2123
 pchemm 3066
 pchemv 3034
 pcher 3038
 pcher2 3040
 pcher2k 3071
 pcherk 3069
 pcnrm2 3017
 pcscal 3018
 pcsscal 3018
 pcswap 3019
 pcsymm 3073
 pcsyr2k 3079
 pcsyrk 3076
 pctradd 3061
 pctranu 3083
 pctrmm 3086
 pctrmv 3051
 pctrsm 3089
 pctrvsv 3056
 pdagemm 3025
 pdamax 3006
 pdasum 3007
 pdasymv 3044
 pdatrmv 3053
 pdaxpy 3009
 pdcopy 3010
 pddot 3012
 PDE support 3093
 pdgeadd 3059
 pdgebal 2179
 pdgecon 1912
 pdgemm 3063
 pdgemv 3022
 pdger 3028
 pdhseqr 2047
 pdlaiectb 2216
 pdlaiectl 2216
 pdlamve 2217
 pdlaqr0 2233
 pdlaqr1 2235
 pdlaqr2 2238
 pdlaqr3 2241
 pdlaqr4 2244
 pdlaqr5 2247
 pdnrm2 3017
 pdrot 2341
 pdscal 3018
 pdswap 3019
 pdseyvr 2106
 pdsymm 3073
 pdsymv 3042
 pdsyrr 3047
 pdsyrr2 3048
 pdsyrr2k 3079
 pdsyrk 3076
 pdtradd 3061
 pdtran 3082
 pdtranc 3085
 pdtrmm 3086
 pdtrmv 3051
 pdtrord 2351
 pdtrsen 2355
 pdtrsm 3089
 pdtrsv 3056
 pdzasum 3007
 permutation matrix 3348
 picopy 3010

pivoting matrix rows or columns 2227
 planar rotation 2341
 pmpcol 2168
 pmpim2 2169
 points rotation
 in the modified plane 70
 in the plane 67
 Poisson 2815
 Poisson problem
 on a sphere 3113
 three-dimensional 3116
 two-dimensional 3112
 Poisson Solver
 routines
 ?_commit_Helmholtz_2D 3122
 ?_commit_Helmholtz_3D 3122
 ?_commit_sph_np 3130
 ?_commit_sph_p 3130
 ?_Helmholtz_2D 3125
 ?_Helmholtz_3D 3125
 ?_init_Helmholtz_2D 3119
 ?_init_Helmholtz_3D 3119
 ?_init_sph_np 3129
 ?_init_sph_p 3129
 ?_sph_np 3132
 ?_sph_p 3132
 free_Helmholtz_2D 3128
 free_Helmholtz_3D 3128
 free_sph_np 3134
 free_sph_p 3134
 structure 3110
 Poisson Solver Interface 3110
 PoissonV 2817
 pprfs 570
 pptrs 472
 preconditioned Jacobi SVD 1238
 preconditioners based on incomplete LU factorization
 dcsrilu0 2530
 dcsrilut 2533
 Preconditioners Interface Description 2529
 process grid 1873, 3003
 product
 distributed matrix-vector
 general matrix 3022, 3025
 distributed vector-scalar 3018
 matrix-vector
 distributed Hermitian matrix 3034, 3036
 distributed symmetric matrix 3042, 3044
 distributed triangular matrix 3051, 3053
 general matrix 81, 84, 390, 392, 1788
 Hermitian indefinite matrix 1800
 Hermitian matrix 93, 96, 102
 real symmetric matrix 111, 117
 symmetric indefinite matrix 1832
 triangular matrix 123, 129, 134
 scalar-matrix
 general distributed matrix 3063
 general matrix 139, 395
 Hermitian distributed matrix 3066
 Hermitian matrix 143
 scalar-matrix-matrix
 general distributed matrix 3063
 general matrix 139, 395
 Hermitian distributed matrix 3066
 Hermitian matrix 143
 symmetric distributed matrix 3073
 symmetric matrix 151
 triangular distributed matrix 3086
 triangular matrix 161
 vector-scalar 74

product:matrix-vector	pzgemm 3063
general matrix	pzgemv 3022
band storage 81	pzgerc 3030
Hermitian matrix	pzgeru 3032
band storage 93	pzhheevr 2123
packed storage 102	pzhemm 3066
real symmetric matrix	pzhemv 3034
packed storage 111	pzher 3038
symmetric matrix	pzher2 3040
band storage 108	pzher2k 3071
triangular matrix	pzhherk 3069
band storage 123	pznrm2 3017
packed storage 129	pzscal 3018
psagemv 3025	pzswap 3019
psamax 3006	pzsymm 3073
psasum 3007	pzsyr2k 3079
psasymv 3044	pzsyrk 3076
psatrmv 3053	pztradd 3061
psaxpy 3009	pztranu 3083
pscasmus 3007	pztrmm 3086
pscscopy 3010	pztrmv 3051
psdot 3012	pztrsm 3089
pseudorandom numbers 2725	pztrsv 3056
psgeadd 3059	
psgebal 2179	
psgecon 1912	
psgemm 3063	
psgemv 3022	
psger 3028	
pshseqr 2047	
pslaiect 2216	
pslamve 2217	
pslaqr0 2233	
pslaqr1 2235	
pslaqr2 2238	
pslaqr3 2241	
pslaqr4 2244	
pslaqr5 2247	
psnrm2 3017	
psrot 2341	
psscal 3018	
psswap 3019	
pssyevr 2106	
pssymm 3073	
pssymv 3042	
pssyr 3047	
pssyr2 3048	
pssyr2k 3079	
pssyrk 3076	
pstradd 3061	
pstran 3082	
pstranc 3085	
pstrmm 3086	
pstrmv 3051	
pstrord 2351	
pstrsen 2355	
pstrsm 3089	
pstrsv 3056	
pxerbla 3199	
pzagemv 3025	
pzahemv 3036	
pzamax 3006	
pzatrmv 3053	
pzaxpy 3009	
pzcscopy 3010	
pzdottedc 3014	
pzdottedu 3015	
pzdscal 3018	
psgeadd 3059	
psgecon 1912	
	Q
	QL factorization
	computing the elements of
	complex matrix Q 838
	orthogonal matrix Q 1973
	real matrix Q 836
	unitary matrix Q 1975
	general rectangular matrix
	LAPACK 1391
	ScalAPACK 2190
	multiplying general matrix by
	orthogonal matrix Q 1977
	unitary matrix Q 1980
	QR factorization
	applying matrix obtained from block reflector to
	general matrix
	orthogonal/unitary matrix Q 874
	computing the elements of
	orthogonal matrix Q 812, 1948
	unitary matrix Q 816, 1950
	general rectangular matrix
	LAPACK 1393, 1394, 1400
	ScalAPACK 2192, 2194
	multiplying general matrix by
	orthogonal/unitary matrix Q 804
	with pivoting
	ScalAPACK 1946
	quasi-random numbers 2725
	quasi-triangular matrix
	LAPACK 990, 1041
	ScalAPACK 2035
	quasi-triangular system of equations 1538
	R
	random number generators 2725
	random stream 2734
	random stream descriptor 2727
	Random Streams 2734
	rank-1 update
	conjugated, distributed general matrix 3030
	conjugated, general matrix 89
	distributed general matrix 3028
	distributed Hermitian matrix 3038

distributed symmetric matrix 3047
 general matrix 87
 Hermitian matrix
 packed storage 104
 real symmetric matrix
 packed storage 113
 unconjugated, distributed general matrix 3032
 unconjugated, general matrix 91
 rank-2 update
 distributed Hermitian matrix 3040
 distributed symmetric matrix 3048
 Hermitian matrix
 packed storage 106
 symmetric matrix
 packed storage 115
 rank-2k update
 Hermitian distributed matrix 3071
 Hermitian matrix 148
 symmetric distributed matrix 3079
 symmetric matrix 158
 rank-k update
 distributed Hermitian matrix 3069
 Hermitian matrix 146
 symmetric distributed matrix 3076
 rank-n update
 symmetric matrix 155
 Rayleigh 2787
 RCI CG Interface 2498
 RCI CG sparse solver routines
 dcg 2513, 2518
 dcg_check 2512
 dcg_get 2515
 dcg_init 2511
 dcgmrhs_check 2517
 dcgmrhs_get 2520
 dcgmrhs_init 2516
 RCI FGMRES Interface 2503
 RCI FGMRES sparse solver routines
 dfgmres_check 2522
 dfgmres_get 2525
 dfgmres_init 2521
 RCI GMRES sparse solver routines
 dfgres 2523
 RCI ISS 2496
 RCI ISS interface 2496
 RCI ISS sparse solver routines
 implementation details 2526
 real matrix
 QR factorization
 with pivoting 1514
 real symmetric matrix
 1-norm value 1493
 Frobenius norm 1493
 infinity- norm 1493
 largest absolute value of element 1493
 real symmetric tridiagonal matrix
 1-norm value 1492
 Frobenius norm 1492
 infinity- norm 1492
 largest absolute value of element 1492
 reducing generalized eigenvalue problems
 LAPACK 975
 ScaLAPACK 2063
 reduction to upper Hessenberg form
 general matrix 2184
 general square matrix 1388
 refining solutions of linear equations
 band matrix 547
 banded matrix 550, 1781, 1818
 general matrix 537, 540, 1794, 1921
 Hermitian indefinite matrix 591, 1805
 Hermitian matrix
 packed storage 600
 Hermitian positive-definite matrix
 band storage 573
 packed storage 570
 symmetric indefinite matrix 582, 1839
 symmetric matrix
 packed storage 597
 symmetric positive-definite matrix
 band storage 573
 packed storage 570
 symmetric/Hermitian positive-definite distributed matrix 1925
 tridiagonal matrix 557
 RegisterBrgn 2822
 registering a basic generator 2821
 Relatively robust representation (RRR) 2386
 reordering of matrices 3349
 Reverse Communication Interface 2496
 rotation
 of points in the modified plane 70
 of points in the plane 67
 of sparse vectors 177
 parameters for a Givens rotation 68
 parameters of modified Givens transformation 72
 rotation, planar 2341
 routine name conventions
 BLAS 51
 Nonlinear Optimization Solvers 3154
 PBLAS 3004
 Sparse BLAS Level 1 168
 Sparse BLAS Level 2 181
 Sparse BLAS Level 3 181
 RQ factorization
 computing the elements of
 complex matrix Q 849
 orthogonal matrix Q 1986
 real matrix Q 848
 unitary matrix Q 1988

S

SaveStreamF 2751
 SaveStreamM 2754
 sbbscd 1091
 sbdsdc 902
 ScaLAPACK 1873
 ScaLAPACK routines
 1D array redistribution 2251, 2253
 auxiliary routines
 ?combamax1 2170
 ?dbtf2 2408
 ?dbtrf 2410
 ?dttrf 2412
 ?dttrs 2413
 ?lamsh 2361
 ?laref 2369
 ?lasorte 2393
 ?lasrt2 2394
 ?ptrsv 2415
 ?stein2 2406
 ?steqr2 2416
 bdlaapp 2160
 bdlaexc 2162
 bdtrexc 2163
 bslaapp 2160
 bslaexc 2162
 bstrexc 2163

dlaqr6 2363
 dlar1va 2366
 dlarrb2 2371
 dlarrd2 2374
 dlarre2 2377
 dlarre2a 2381
 dlarrf2 2386
 dlarrv2 2388
 dstegr2 2395
 dstegr2a 2399
 dstegr2b 2402
 p?dbtrsv 2172
 p?dttrsv 2175
 p?gebd2 2181
 p?gehd2 2184
 p?gelq2 2187
 p?geql2 2190
 p?geqr2 2192
 p?gerq2 2194
 p?getf2 2197
 p?labrd 2199
 p?lacgv 2166
 p?lacon 2203
 p?laconsb 2205
 p?lacp2 2206
 p?lacp3 2208
 p?lacpy 2209
 p?laevswp 2211
 p?lahrd 2213
 p?lalect 2216
 p?lange 2219
 p?lanhs 2221
 p?lansy, p?lanhe 2223
 p?lantr 2225
 p?lapiv 2227
 p?laqge 2230
 p?laqsy 2249
 p?lared1d 2251
 p?lared2d 2253
 p?larf 2254
 p?larfb 2257
 p?larfc 2261
 p?larfg 2264
 p?larft 2266
 p?larz 2269
 p?larzb 2273
 p?larzc 2277
 p?larzt 2280
 p?lascl 2283
 p?laset 2285
 p?lasmsub 2287
 p?lassq 2288
 p?laswp 2290
 p?latra 2292
 p?latrd 2293
 p?latrs 2297
 p?latrz 2300
 p?lauu2 2302
 p?lauum 2304
 p?lawil 2305
 p?max1 2167
 p?org2l/p?ung2l 2307
 p?org2r/p?ung2r 2309
 p?orgl2/p?ungl2 2312
 p?orgr2/p?ungr2 2314
 p?orm2l/p?unm2l 2317
 p?orm2r/p?unm2r 2320
 p?orml2/p?unml2 2324
 p?ormr2/p?unmr2 2328
 p?pbtrsv 2332
 p?potf2 2340
 p?pttrsv 2336
 p?rscl 2344
 p?sum1 2171
 p?srgs2/p?hegs2 2345
 p?sytd2/p?hetd2 2348
 p?trti2 2359
 pdgebal 2179
 pdhseqr 2047
 pdlaiectb 2216
 pdlaiectl 2216
 pdlamve 2217
 pdlaqr0 2233
 pdlaqr1 2235
 pdlaqr2 2238
 pdlaqr3 2241
 pdlaqr4 2244
 pdlaqr5 2247
 pdrot 2341
 pdtrord 2351
 pdtrsen 2355
 pmpcol 2168
 pmpim2 2169
 psgebal 2179
 pshseqr 2047
 pslaiect 2216
 pslamve 2217
 pslaqr0 2233
 pslaqr1 2235
 pslaqr2 2238
 pslaqr3 2241
 pslaqr4 2244
 pslaqr5 2247
 psrot 2341
 pstrord 2351
 pstrsen 2355
 slaqr6 2363
 slar1va 2366
 slarrb2 2371
 slarrd2 2374
 slarre2 2377
 slarre2a 2381
 slarrf2 2386
 slarrv2 2388
 sssteqr2 2395
 sssteqr2a 2399
 sssteqr2b 2402
 block reflector
 triangular factor 2266, 2280
 Cholesky factorization 1890
 complex matrix
 complex elementary reflector 2277
 complex vector
 1-norm using true absolute value
 2171
 complex vector conjugation 2166
 condition number estimation
 p?gecon 1912
 p?pocon 1915
 p?trcon 1918
 copying matrices
 p?gemr2d 2423
 p?trmr2d 2425
 driver routines
 p?dbsv 2078
 p?dtsv 2080
 p?gbsv 2075
 p?gels 2097
 p?gesv 2067
 p?gesvd 2135

p?gesvx 2069
 p?heev 2116
 p?heevd 2120
 p?heevx 2128
 p?hegvx 2146
 p?pbsv 2091
 p?posv 2083
 p?posvx 2085
 p?ptsv 2094
 p?syev 2100
 p?syevd 2103
 p?syevx 2110
 p?sygsvx 2139
 pccheevr 2123
 pdssyevr 2106
 pssyevr 2106
 pzheevr 2123
 error estimation
 p?ttrfs 1928
 error handling
 pxerbla 3199
 general matrix
 block reflector 2273
 elementary reflector 2269
 LU factorization 2197
 reduction to upper Hessenberg
 form 2184
 general rectangular matrix
 elementary reflector 2254
 LQ factorization 2187
 QL factorization 2190
 QR factorization 2192
 reduction to bidiagonal form 2199
 reduction to real bidiagonal form
 2181
 row interchanges 2290
 RQ factorization 2194
 generalized eigenvalue problems
 p?hegst 2065
 p?sygst 2063
 Householder matrix
 elementary reflector 2264
 LQ factorization
 p?gelq2 2187
 p?gelqf 1958
 p?orglq 1961
 p?ormlq 1965
 p?unqlq 1963
 p?unmlq 1968
 LU factorization
 p?dbtrsv 2172
 p?dttrf 1884
 p?dttrsv 2175
 p?getf2 2197
 matrix equilibration
 p?geequ 1938
 p?poequ 1940
 matrix inversion
 p?getri 1932
 p?potri 1935
 p?trtri 1936
 nonsymmetric eigenvalue problems
 p?gehrd 2035
 p?lahqr 2045
 p?ormhr 2038
 p?unmhr 2042
 QL factorization
 ?geqlf 1971
 ?ungql 1975
 p?geql2 2190
 p?orgql 1973
 p?ormql 1977
 p?unmql 1980
 QR factorization
 p?geqpf 1946
 p?geqr2 2192
 p?ggqr2 2005
 p?orgqr 1948
 p?ormqr 1952
 p?ungqr 1950
 p?unmqr 1955
 RQ factorization
 p?gerq2 2194
 p?gerqf 1983
 p?ggrqf 2009
 p?orgrq 1986
 p?ormrq 1989
 p?ungrq 1988
 p?unmrq 1992
 RZ factorization
 p?ormrz 1998
 p?tzrzf 1996
 p?unmrz 2002
 singular value decomposition
 p?gebrd 2050
 p?ormbr 2054
 p?unmbr 2058
 solution refinement and error
 estimation
 p?gerfs 1921
 p?porfs 1925
 solving linear equations
 ?dttrsv 2413
 ?pttrsv 2415
 p?dbtrs 1898
 p?dttrs 1900
 p?gbtrs 1895
 p?getrs 1893
 p?potrs 1903
 p?ptrs 1907
 p?trtrs 1909
 symmetric eigenproblems
 p?hetrd 2021
 p?ormtr 2017
 p?stebz 2027
 p?stein 2031
 p?sytrd 2014
 p?unmtr 2024
 symmetric eigenvalue problems
 ?stein2 2406
 ?steqr2 2416
 trapezoidal matrix 2300
 triangular factorization
 ?dbtrf 2410
 ?dttrf 2412
 p?dbtrsv 2172
 p?dttrsv 2175
 p?gbtrf 1879
 p?getrf 1877
 p?pbtrf 1888
 p?potrf 1886
 p?pttrf 1890
 triangular system of equations 2297
 updating sum of squares 2288
 utility functions and routines
 p?labad 2419
 p?lachkieee 2420
 p?lamch 2421
 p?lasnbt 2422
 pxerbla 3199

scalar-matrix product 139, 143, 151, 395, 3063, 3066, 3073
scalar-matrix-matrix product
 general distributed matrix 3063
 general matrix 139, 395
 symmetric distributed matrix 3073
 symmetric matrix 151
 triangular distributed matrix 3086
 triangular matrix 161
scaling
 general rectangular matrix 2230
 symmetric/Hermitian matrix 2249
scaling factors
 general rectangular distributed matrix 1938
 Hermitian positive definite distributed matrix 1940
 symmetric positive definite distributed matrix 1940
scattering compressed sparse vector's elements into full storage form 179
Schur decomposition 1061, 1064, 2233, 2244
Schur factorization
 reordering 2355
scsum1 1384
second/dsecnd 3203
Service Functions 2578
Service Routines 2738
SetInternalDecimation 2849
sgbcon 504
sgbrfsx 550
sgbsvx 686
sgbtrs 461
sgecon 502
sgejsv 1238
sgeqpf 806
sgesvj 1245
sgtrfs 557
shgeqz 1050
shseqr 1011
simple driver 1875
Single Dynamic Library
 mkl_set_exit_handler 3200
 mkl_set_interface_layer 3219
 mkl_set_progress 3222
 mkl_set_threading_layer 3220
 mkl_set_xerbla 3221
single node matrix 2361
singular value decomposition
 LAPACK 878
 LAPACK routines, singular value decomposition 2050
 ScalAPACK 2050, 2135
 See also LAPACK routines, singular value decomposition 878
Singular Value Decomposition 1229
sjacobi 3175
sjacobi_delete 3175
sjacobi_init 3172
sjacobi_solve 3173
sjacobix 3177
SkipAheadStream 2760
sla_gbavm 1773
sla_gbrcond 1775
sla_gbrfsx_extended 1781
sla_gbrpvgrw 1787
sla_geamv 1788
sla_gercond 1790
sla_gerfsx_extended 1794
sla_gerpvgrw 1829
sla_lin_berr 1812
sla_porcond 1813
sla_porfsx_extended 1818
sla_porpvgrw 1824

sla_syamv 1832
sla_syrcond 1834
sla_syrfsx_extended 1839
sla_syrpvgrw 1845
sla_wwaddw 1847
slag2d 1734
slapmr 1504
slapmt 1505
slaqr6 2363
slar1va 2366
slarfb 1547
slarf1 1553
slarrb2 2371
slarrd2 2374
slarre2 2377
slarre2a 2381
slarrf2 2386
slarrv2 2388
slarscl2 1830
slartgp 1583
slartgs 1584
slascl2 1831
slatps 1659
slatrd 1661
slatrs 1663
slatzr 1667
slauu2 1669
slauum 1670
small subdiagonal element 2287
smallest absolute value of a vector element 78
sNewAbstractStream 2746
solver
 direct 3347
 iterative 3347
Solver
 Sparse 2429
solving linear equations 461
solving linear equations. linear equations 1895
solving linear equations. See linear equations 1468
sorbdb 1098
sorbdb1 1671
sorbdb2 1674
sorbdb3 1678
sorbdb4 1681
sorbdb5 1684
sorbdb6 1687
sorcisd 1255
sorcisd2by1 1261
sorg2l 1689
sorg2r 1691
sorgl2 1692
sorgr2 1694
sorm2l 1695
sorm2r 1697
sorml2 1699
sormr2 1701
sormr3 1703
sorting
 eigenpairs 2393
 numbers in increasing/decreasing order
 LAPACK 1640
 ScalAPACK 2394
Sparse BLAS Level 1
 data types 168
 naming conventions 168
Sparse BLAS Level 1 routines and functions
 ?axpyi 169
 ?dotci 172
 ?doti 171
 ?dotui 173

?gthr 175
?gthrz 176
?roti 177
?sctr 179
Sparse BLAS Level 2
 naming conventions 181
sparse BLAS Level 2 routines
 mkl_?bsrgemv 194
 mkl_?bsrmv 259
 mkl_?bsrsv 276
 mkl_?bsrsymv 206
 mkl_?bsrtrsv 218
 mkl_?coogemv 197
 mkl_?coomv 268
 mkl_?coosv 284
 mkl_?coosymv 209
 mkl_?cootrv 221
 mkl_?cscmv 264
 mkl_?cscsv 280
 mkl_?csrgemv 191
 mkl_?csrsv 255
 mkl_?csrsv 272
 mkl_?csrsymv 203
 mkl_?csrtrsv 215
 mkl_?diagemv 200
 mkl_?diamv 322
 mkl_?diasv 330
 mkl_?diasymv 212
 mkl_?diatrv 224
 mkl_?skymv 326
 mkl_?skysv 334
 mkl_cspblas_?bsrgemv 231
 mkl_cspblas_?bsrsymv 240
 mkl_cspblas_?bsrtrsv 249
 mkl_cspblas_?coogemv 234
 mkl_cspblas_?coosymv 243
 mkl_cspblas_?cootrv 252
 mkl_cspblas_?csrgemv 228
 mkl_cspblas_?csrsymv 237
 mkl_cspblas_?csrtrsv 246
Sparse BLAS Level 3
 naming conventions 181
sparse BLAS Level 3 routines
 mkl_?bsrmm 292
 mkl_?bsrsm 318
 mkl_?coomm 302
 mkl_?coosm 315
 mkl_?cscmm 297
 mkl_?cscsm 310
 mkl_?csradd 375
 mkl_?csrmm 288
 mkl_?csrmultcsr 379
 mkl_?csrmultd 384
 mkl_?csrsm 306
 mkl_?diamm 337
 mkl_?diasm 346
 mkl_?skymm 341
 mkl_?skysm 350
sparse BLAS routines
 mkl_?csrbsr 360
 mkl_?csrcoo 357
 mkl_?csrcsc 364
 mkl_?csrdia 367
 mkl_?csrsky 371
 mkl_?dnsCSR 353
sparse matrices 180
sparse matrix 180
Sparse Matrix Checker Routines 2537
Sparse Matrix Storage Formats 182
sparse solver
parallel direct sparse solver interface
 mkl_pardiso_pivot 2444
 pardiso 2434
 pardiso_64 2442
 pardiso_getdiag 2445
 pardiso_getenv 2443
 pardiso_handle_delete 2449
 pardiso_handle_delete_64 2452
 pardiso_handle_restore 2448
 pardiso_handle_restore_64 2451
 pardiso_handle_store 2447
 pardiso_handle_store_64 2450
 pardiso_setenv 2443
 pardisoinit 2440
parallel direct sparse solver interface for clusters
 cluster_sparse_solver 2469
Sparse Solver
 direct sparse solver interface
 dss_create 2482
 dss_define_structure
 dss_define_structure 2483
 dss_delete 2492
 dss_factor 2487
 dss_factor_complex 2487
 dss_factor_real 2487
 dss_reorder 2485
 dss_solve 2489
 dss_solve_complex 2489
 dss_solve_real 2489
 dss_statistics 2493
 mkl_cvt_to_null_terminated_str 2496
 iterative sparse solver interface
 dcg 2513
 dcg_check 2512
 dcg_get 2515
 dcg_init 2511
 dcgmrhs 2518
 dcgmrhs_check 2517
 dcgmrhs_get 2520
 dcgmrhs_init 2516
 dfgmres 2523
 dfgmres_check 2522
 dfgmres_get 2525
 dfgmres_init 2521
 preconditioners based on incomplete LU
 factorization
 dcsrilu0 2530
 dcsrilut 2533
Sparse Solvers 2429, 2452, 2457, 2467, 2474, 2543,
 2544, 2546, 2550, 2556
sparse vectors
 adding and scaling 169
 complex dot product, conjugated 172
 complex dot product, unconjugated 173
 compressed form 167
 converting to compressed form 175, 176
 converting to full-storage form 179
 full-storage form 167
 Givens rotation 177
 norm 168
 passed to BLAS level 1 routines 168
 real dot product 171
 scaling 168
sparse_matrix_checker 2537
sparse_matrix_init 2539
spbtf2 1706
specific hardware support
 mkl_enable_instructions 3248
Spline Methods 3323
split Cholesky factorization (band matrices) 989

sporfsx 563
 spotf2 1707
 spprfs 570
 spptrs 472
 sptts2 1708
 square matrix
 1-norm estimation
 LAPACK 1410, 1412
 ScalAPACK 2203
 srscl 1710
 ssteqr2 2395
 ssteqr2a 2399
 ssteqr2b 2402
 ssyconv 1379
 ssygs2 1715
 ssyswapr 1711
 ssyswapr1 1714
 ssytd2 1717
 ssytf2 1719
 ssytf2_rook 1721
 ssytri2 626
 ssytri2x 630
 ssytrs2 486
 stgex2 1725
 stgsy2 1727
 stream 2734
 strexc 1031
 stride. increment 3365
 strnlsp_check 3157
 strnlsp_delete 3162
 strnlsp_get 3161
 strnlsp_init 3155
 strnlsp_solve 3159
 strnlspbc_check 3165
 strnlspbc_delete 3171
 strnlspbc_get 3170
 strnlspbc_init 3163
 strnlspbc_solve 3168
 strti2 1731
 sum
 of distributed vectors 3009
 of magnitudes of elements of a distributed vector
 3007
 of magnitudes of the vector elements 56
 of sparse vector and full-storage vector 169
 of vectors 57, 388
 sum of squares
 updating
 LAPACK 1641
 ScalAPACK 2288
 summary statistics
 vslscompute 2921
 vslSSCompute 2921
 vslsseditcorparameterization 2916
 vslSSEditCorParameterization 2916
 vslsseditcovcor 2897
 vslSSEditCovCor 2897
 vslsseditcp 2899
 vslSSEditCP 2899
 vslsseditmissingvalues 2912
 vslSSEditMissingValues 2912
 vslsseditmoments 2893
 vslSSEditMoments 2893
 vslsseditoutliersdetection 2910
 vslSSEditOutliersDetection 2910
 vslsseditpartialcovcor 2901
 vslSSEditPartialCovCor 2901
 vslsseditpooledcovariance 2906
 vslSSEditPooledCovariance 2906
 vslsseditquantiles 2903
 vslSSEditQuantiles 2903
 vslsseditrobustcovariance 2908
 vslSSEditRobustCovariance 2908
 vslsseditstreamquantiles 2904
 vslSSEditStreamQuantiles 2904
 vslsseditsums 2895
 vslSSEditSums 2895
 vslssedittask 2884
 vslSSEditTask 2884
 vslssnewtask 2881
 vslSSNewTask 2881
 vslssedittask 2884
 vslSSEditTask 2884
 vslssdeletetask 2922
 vslSSDeleteTask 2922
 vslsscompute 2921
 vslSSCompute 2921
 vslsseditcorparameterization 2916
 vslSSEditCorParameterization 2916
 vslsseditcovcor 2897
 vslSSEditCovCor 2897
 vslsseditcp 2899
 vslSSEditCP 2899
 vslsseditmissingvalues 2912
 vslSSEditMissingValues 2912
 vslsseditmoments 2893
 vslSSEditMoments 2893
 vslsseditoutliersdetection 2910
 vslSSEditOutliersDetection 2910
 vslsseditpartialcovcor 2901
 vslSSEditPartialCovCor 2901
 vslsseditpooledcovariance 2906
 vslSSEditPooledCovariance 2906
 vslsseditquantiles 2903
 vslSSEditQuantiles 2903
 vslsseditrobustcovariance 2908
 vslSSEditRobustCovariance 2908
 vslsseditstreamquantiles 2904
 vslSSEditStreamQuantiles 2904
 vslsseditsums 2895
 vslSSEditSums 2895
 vslssedittask 2884
 vslSSEditTask 2884
 vslssnewtask 2881
 vslSSNewTask 2881
Summary Statistics 2874
 summary statistics usage examples 2923
 support functions
 mkl_calloc 3213
 mkl_enable_instructions 3248
 mkl_free 3215
 mkl_malloc 3212
 mkl_mem_stat 3210
 mkl_mic_disable 3225
 mkl_mic_enable 3224
 mkl_mic_free_memory 3231
 mkl_mic_get_device_count 3225
 mkl_mic_get_resource_limit 3238
 mkl_mic_get_workdivision 3228
 mkl_mic_register_memory 3232
 mkl_mic_set_device_num_threads 3233
 mkl_mic_set_max_memory 3229
 mkl_mic_set_offload_report 3239
 mkl_mic_set_resource_limit 3235
 mkl_mic_set_workdivision 3226
 mkl_peak_mem_usage 3211
 mkl_progress 3246
 mkl_realloc 3214
 mkl_set_env_mode 3249
 mkl_verbose 3250

Support Functions
 exception handling 3198
 handling fatal errors 3200
 support functions for CNR
 mkl_cbwr_get 3241
 mkl_cbwr_get_auto_branch 3242
 mkl_cbwr_set 3240
 support routines
 mkl_disable_fast_mm 3209
 mkl_free_buffers 3207
 mkl_thread_free_buffers 3208
 progress information 3246
 SVD (singular value decomposition)
 LAPACK 878
 ScalAPACK 2050
 swapping adjacent diagonal blocks 1446, 1725
 swapping distributed vectors 3019
 swapping vectors 75
 Sylvester's equation 1038
 symmetric band matrix
 1-norm value 1486
 Frobenius norm 1486
 infinity- norm 1486
 largest absolute value of element 1486
 symmetric distributed matrix
 rank-n update 3076, 3079
 scalar-matrix-matrix product 3073
 Symmetric Eigenproblems 1124
 symmetric indefinite matrix
 factorization with diagonal pivoting method 1719, 1721
 matrix-vector product 1832
 symmetric matrix
 Bunch-Kaufman factorization
 packed storage 454
 eigenvalues and eigenvectors 2100, 2103, 2106, 2110
 estimating the condition number
 packed storage 526
 generalized eigenvalue problems 975
 inverting the matrix
 packed storage 634
 matrix-vector product
 band storage 108
 packed storage 111, 1376
 rank-1 update
 packed storage 113, 1377
 rank-2 update
 packed storage 115
 rank-2k update 158
 rank-n update 155
 reducing to standard form
 LAPACK 1715
 ScalAPACK 2345
 reducing to tridiagonal form
 LAPACK 1661
 ScalAPACK 2293
 scalar-matrix-matrix product 151
 scaling 2249
 solving systems of linear equations
 packed storage 490
 symmetric matrix in packed form
 1-norm value 1489
 Frobenius norm 1489
 infinity- norm 1489
 largest absolute value of element 1489
 symmetric positive definite distributed matrix
 computing scaling factors 1940
 equilibration 1940
 symmetric positive semidefinite matrix
 Cholesky factorization 432
 symmetric positive-definite band matrix
 Cholesky factorization 1706
 symmetric positive-definite distributed matrix
 inverting the matrix 1935
 symmetric positive-definite matrix
 Cholesky factorization
 band storage 438, 1888
 LAPACK 1707
 packed storage 436
 ScalAPACK 1886, 2340
 estimating the condition number
 band storage 513
 packed storage 511
 tridiagonal matrix 516
 inverting the matrix
 packed storage 617
 solving systems of linear equations
 band storage 474, 1905
 LAPACK 468
 packed storage 472
 ScalAPACK 1903
 symmetric positive-definite tridiagonal matrix
 solving systems of linear equations 1907
 symmetric tridiagonal matrix
 eigenvalues and eigenvectors 2374
 system of linear equations
 with a distributed triangular matrix 3056
 with a triangular matrix
 band storage 126
 packed storage 131
 systems of linear equations
 linear equations 2413
 systems of linear equations/linear equations 1893
 syswapr 1711
 syswapr1 1714
 sytri2 626
 sytri2x 630

T

Task Computation Routines 3323
 Task Creation and Initialization
 NewTask1d 3305
 Task Status 3303
 threading control
 mkl_domain_get_max_threads 3195
 mkl_domain_set_num_threads 3189
 mkl_get_dynamic 3196
 mkl_get_max_threads 3194
 mkl_set_dynamic 3193
 mkl_set_num_threads 3188
 mkl_set_num_threads_local 3191
 Threading Control 3184, 3187
 timing functions
 mkl_get_clocks_frequency 3206
 MKL_Get_Cpu_Clocks 3204
 mkl_get_cpu_frequency 3205
 mkl_get_max_cpu_frequency 3206
 second/dsecnd 3203
 TR routines
 ?trnlsp_check 3157
 ?trnlsp_delete 3162
 ?trnlsp_get 3161
 ?trnlsp_init 3155
 ?trnlsp_solve 3159
 ?trnlspbc_check 3165
 ?trnlspbc_delete 3171
 ?trnlspbc_get 3170

?trnlspbc_init 3163
?trnlspbc_solve 3168
nonlinear least squares problem
 with linear bound constraints 3163
 without constraints 3154
organization and implementation 3153
transposition
 distributed complex matrix 3083
 distributed complex matrix, conjugated 3085
 distributed real matrix 3082
Transposition and General Memory Movement Routines
 387
transposition parameter 3368
trapezoidal matrix
 1-norm value 1500
 copying 2425
 Frobenius norm 1500
 infinity- norm 1500
 largest absolute value of element 1500
 reduction to triangular form 2300
 RZ factorization
 LAPACK 856
 ScalAPACK 1996
trexc 1031
triangular band matrix
 1-norm value 1496
 Frobenius norm 1496
 infinity- norm 1496
 largest absolute value of element 1496
triangular banded equations
 LAPACK 1655
 ScalAPACK 2332
triangular distributed matrix
 inverting the matrix 1936
 scalar-matrix-matrix product 3086
triangular factorization
 band matrix 424, 1879, 1882, 2172, 2410
 diagonally dominant tridiagonal matrix
 LAPACK 428
 general matrix 421, 1877
 Hermitian matrix
 packed storage 457
 Hermitian positive semidefinite matrix 432
 Hermitian positive-definite matrix
 band storage 438, 1888
 packed storage 436
 tridiagonal matrix 440, 1890
 symmetric matrix
 packed storage 454
 symmetric positive semidefinite matrix 432
 symmetric positive-definite matrix
 band storage 438, 1888
 packed storage 436
 tridiagonal matrix 440, 1890
 tridiagonal matrix
 LAPACK 426
 ScalAPACK 2412
triangular matrix
 1-norm value
 LAPACK 1500
 ScalAPACK 2225
 copying 1753, 1754, 1762, 1764, 1765, 1767
 estimating the condition number
 band storage 534
 packed storage 532
 Frobenius norm
 LAPACK 1500
 ScalAPACK 2225
 infinity- norm
 LAPACK 1500
ScalAPACK 2225
inverting the matrix
 LAPACK 1731
 packed storage 641
 ScalAPACK 2359
largest absolute value of element
 LAPACK 1500
 ScalAPACK 2225
matrix-vector product
 band storage 123
 packed storage 129
product
 blocked algorithm 1670, 2304
 LAPACK 1669, 1670
 ScalAPACK 2302, 2304
 unblocked algorithm 1669
ScalAPACK 2035
scalar-matrix-matrix product 161
 solving systems of linear equations
 band storage 126, 499
 packed storage 131, 497
 ScalAPACK 1909
 swapping adjacent diagonal blocks 1725
triangular matrix factorization
 Hermitian positive-definite matrix 430
 symmetric positive-definite matrix 430
triangular matrix in packed form
 1-norm value 1498
 Frobenius norm 1498
 infinity- norm 1498
 largest absolute value of element 1498
triangular pentagonal matrix
 QR factorization 872, 1756, 1759
triangular system of equations
 solving with scale factor
 LAPACK 1663
 ScalAPACK 2297
tridiagonal matrix
 base representations and eigenvalues 2377, 2381
 eigenvalues and eigenvectors 2395, 2402
 eigenvectors 2388
 estimating the condition number 506
 solving systems of linear equations
 ScalAPACK 2413
tridiagonal system of equations 1708
tridiagonal triangular factorization
 band matrix 2175
tridiagonal triangular system of equations 2336
trigonometric transform
 backward cosine 3095
 backward sine 3094
 backward staggered cosine 3095
 backward staggered sine 3094
 backward twice staggered cosine 3095
 backward twice staggered sine 3094
 forward cosine 3094
 forward sine 3094
 forward staggered cosine 3095
 forward staggered sine 3094
 forward twice staggered cosine 3095
 forward twice staggered sine 3094
Trigonometric Transform interface
 routines
 ?_backward_trig_transform 3102
 ?_commit_trig_transform 3098
 ?_forward_trig_transform 3101
 ?_init_trig_transform 3097
 free_trig_transform 3104
Trigonometric Transforms interface 3097
TT interface 3093

TT routines 3097
 two matrices
 QR factorization
 LAPACK 865
 ScalAPACK 2005

U

ungbr 892
 Uniform (continuous) 2767
 Uniform (discrete) 2801
 UniformBits 2803
 UniformBits32 2805
 UniformBits64 2806
 unitary matrix
 CS decomposition
 LAPACK 1091, 1098, 1255, 1261, 1671,
 1674, 1678, 1681, 1684, 1687
 from LQ factorization
 LAPACK 1692
 ScalAPACK 2312
 from QL factorization
 LAPACK 1689, 1695
 ScalAPACK 2307, 2317
 from QR factorization
 LAPACK 1691
 ScalAPACK 2309
 from RQ factorization
 LAPACK 1694
 ScalAPACK 2314
 ScalAPACK 2035, 2050
 Unpack Functions 2578
 updating
 rank-1
 distributed general matrix 3028
 distributed Hermitian matrix 3038
 distributed symmetric matrix 3047
 general matrix 87
 Hermitian matrix 98, 104
 real symmetric matrix 113, 119
 rank-1, conjugated
 distributed general matrix 3030
 general matrix 89
 rank-1, unconjugated
 distributed general matrix 3032
 general matrix 91
 rank-2
 distributed Hermitian matrix 3040
 distributed symmetric matrix 3048
 Hermitian matrix 100, 106
 symmetric matrix 115, 121
 rank-2k
 Hermitian distributed matrix 3071
 Hermitian matrix 148
 symmetric distributed matrix 3079
 symmetric matrix 158
 rank-k
 distributed Hermitian matrix 3069
 Hermitian matrix 146
 symmetric distributed matrix 3076
 rank-n
 symmetric matrix 155
 updating:rank-1
 Hermitian matrix
 packed storage 104
 real symmetric matrix
 packed storage 113
 updating:rank-2
 Hermitian matrix

 packed storage 106
 symmetric matrix
 packed storage 115
 upper Hessenberg matrix
 1-norm value
 LAPACK 1485
 ScalAPACK 2221
 Frobenius norm
 LAPACK 1485
 ScalAPACK 2221
 infinity- norm
 LAPACK 1485
 ScalAPACK 2221
 largest absolute value of element
 LAPACK 1485
 ScalAPACK 2221
 ScalAPACK 2035

V

v?Abs 2595
 v?Acos 2648
 v?Acosh 2667
 v?Add 2583
 v?Arg 2597
 v?Asin 2651
 v?Asinh 2670
 v?Atan 2654
 v?Atan2 2656
 v?Atanh 2673
 v?Cbrt 2610
 v?CdfNorm 2681
 v?CdfNormInv 2688
 v?Ceil 2696
 v?CIS 2644
 v?Conj 2594
 v?Cos 2638
 v?Cosh 2658
 v?Div 2603
 v?Erf 2676
 v?Erfc 2679
 v?ErfcInv 2686
 v?ErfInv 2683
 v?Exp 2625
 v?Expm1 2628
 v?Floor 2694
 v?Frac 2706
 v?Hypot 2623
 v?Inv 2602
 v?InvCbrt 2612
 v?InvSqrt 2609
 v?Igamma 2690
 v?LGamma 2690
 v?LinearFrac 2599
 v?Ln 2630
 v?Log10 2633
 v?Log1p 2636
 v?Modf 2704
 v?Mul 2589
 v?MulByConj 2592
 v?NearbyInt 2700
 v?Pack 2708
 v?Pow 2617
 v?Pow2o3 2613
 v?Pow3o2 2615
 v?Powl 2621
 v?Rint 2702
 v?Round 2699
 v?Sin 2640
 v?SinCos 2642

v?Sinh 2662
v?Sqr 2588
v?Sqrt 2606
v?Sub 2585
v?Tan 2646
v?Tanh 2665
v?tgamma 2692
v?TGamma 2692
v?Trunc 2697
v?Unpack 2711
vcAdd 2583
vcPackI 2708
vcPackM 2708
vcPackV 2708
vcSin 2640
vcSub 2585
vcUnpackI 2711
vcUnpackM 2711
vcUnpackV 2711
vdAdd 2583
vdIgamma 2690
vdLGamma 2690
vdPackI 2708
vdPackM 2708
vdPackV 2708
vdSin 2640
vdSub 2585
vdtgamma 2692
vdTGamma 2692
vdUnpackI 2711
vdUnpackM 2711
vdUnpackV 2711
vector arguments
 array dimension 3365
 default 3366
 examples 3365
 increment 3365
 length 3365
 matrix one-dimensional substructures 3365
 sparse vector 167
vector conjugation 1371, 2166
vector indexing 2579
vector mathematical functions
 absolute value 2595
 addition 2583
 argument 2597
 complementary error function value 2679
 complex exponent of real vector elements 2644
 computing a rounded integer value and raising
 inexact result exception 2702
 computing a rounded integer value in current
 rounding mode 2700
 computing a truncated integer value 2704
 conjugation 2594
 cosine 2638
 cube root 2610
 cumulative normal distribution function value 2681
 denary logarithm 2633
 division 2603
 error function value 2676
 exponential 2625
 exponential of elements decreased by 1 2628
 four-quadrant arctangent 2656
 gamma function 2690, 2692
 hyperbolic cosine 2658
 hyperbolic sine 2662
 hyperbolic tangent 2665
 inverse complementary error function value 2686
 inverse cosine 2648
 inverse cube root 2612
 inverse cumulative normal distribution function value
 2688
 inverse error function value 2683
 inverse hyperbolic cosine 2667
 inverse hyperbolic sine 2670
 inverse hyperbolic tangent 2673
 inverse sine 2651
 inverse square root 2609
 inverse tangent 2654
 inversion 2602
 linear fraction transformation 2599
 multiplication 2589
 multiplication of conjugated vector element 2592
 natural logarithm 2630
 natural logarithm of vector elements increased by 1
 2636
 power 2617
 power (constant) 2621
 power 2/3 2613
 power 3/2 2615
 rounding to nearest integer value 2699
 rounding towards minus infinity 2694, 2706
 rounding towards plus infinity 2696
 rounding towards zero 2697
 scaling 1831
 scaling, reciprocal 1830
 sine 2640
 sine and cosine 2642
 square root 2606
 square root of sum of squares 2623
 squaring 2588
 subtraction 2585
 tangent 2646
Vector Mathematical Functions 2575
vector multiplication
 LAPACK 1710
 ScalAPACK 2344
vector pack function 2708
vector statistics functions
 Bernoulli 2808
 Beta 2798
 Binomial 2811
 Cauchy 2785
 CopyStream 2749
 CopyStreamState 2750
 DeleteStream 2749
 dNewAbstractStream 2744
 Exponential 2777
 Gamma 2796
 Gaussian 2770
 GaussianMV 2772
 Geometric 2809
 GetBrngProperties 2823
 GetNumRegBrngs 2763
 GetStreamSize 2756
 GetStreamStateBrng 2762
 Gumbel 2793
 Hypergeometric 2813
 iNewAbstractStream 2742
 Laplace 2779
 LeapfrogStream 2757
 LoadStreamF 2753
 LoadStreamM 2755
 Lognormal 2790
 NegBinomial 2819
 NewStream 2739
 NewStreamEx 2741
 Poisson 2815
 PoissonV 2817
 Rayleigh 2787

RegisterBrng 2822
 SaveStreamF 2751
 SaveStreamM 2754
 SkipAheadStream 2760
 sNewAbstractStream 2746
 Uniform (continuous) 2767
 Uniform (discrete) 2801
 UniformBits 2803
 UniformBits32 2805
 UniformBits64 2806
 Weibull 2782
 vector unpack function 2711
 vector-scalar product
 sparse vectors 169
 vectors
 adding magnitudes of vector elements 56
 copying 59
 dot product
 complex vectors 64
 complex vectors, conjugated 63
 real vectors 60
 element with the largest absolute value 77
 element with the largest absolute value of real part
 and its index 2170
 element with the smallest absolute value 78
 Euclidean norm 65
 Givens rotation 68
 linear combination of vectors 57, 388
 modified Givens transformation parameters 72
 rotation of points 67
 rotation of points in the modified plane 70
 sparse vectors 168
 sum of vectors 57, 388
 swapping 75
 vector-scalar product 74
 viRngUniformBits 2803
 viRngUniformBits32 2805
 viRngUniformBits64 2806
 vmcAdd 2583
 vmcSin 2640
 vmcSub 2585
 vmdAdd 2583
 vmdSin 2640
 vmdSub 2585
 vml
 Functions Interface 2577
 Input Parameters 2579
 Output Parameters 2579
 VML 2575
 VML arithmetic functions 2583
 VML exponential and logarithmic functions 2625
 VML functions
 mathematical functions
 v?Abs 2595
 v?Acos 2648
 v?Acosh 2667
 v?Add 2583
 v?Arg 2597
 v?Asin 2651
 v?Asinh 2670
 v?Atan 2654
 v?Atan2 2656
 v?Atanh 2673
 v?Cbrt 2610
 v?CdfNorm 2681
 v?CdfNormInv 2688
 v?Ceil 2696
 v?CIS 2644
 v?Conj 2594
 v?Cos 2638
 v?Cosh 2658
 v?Div 2603
 v?Erf 2676
 v?Erfc 2679
 v?ErfcInv 2686
 v?ErfInv 2683
 v?Exp 2625
 v?Expm1 2628
 v?Floor 2694
 v?Frac 2706
 v?Hypot 2623
 v?Inv 2602
 v?InvCbrt 2612
 v?InvSqrt 2609
 v?LGamma 2690
 v?LinearFrac 2599
 v?Ln 2630
 v?Log10 2633
 v?Log1p 2636
 v?Modf 2704
 v?Mul 2589
 v?MulByConj 2592
 v?NearbyInt 2700
 v?Pow 2617
 v?Pow2o3 2613
 v?Pow3o2 2615
 v?Powx 2621
 v?Rint 2702
 v?Round 2699
 v?Sin 2640
 v?SinCos 2642
 v?Sinh 2662
 v?Sqr 2588
 v?Sqrt 2606
 v?Sub 2585
 v?Tan 2646
 v?Tanh 2665
 v?TGamma 2692
 v?Trunc 2697
 pack/unpack functions
 v?Pack 2708
 v?Unpack 2711
 service functions
 ClearErrorCallBack 2722
 ClearErrStatus 2719
 GetErrorCallBack 2722
 GetErrStatus 2718
 GetMode 2716
 MKLFreeTIs 2717
 SetErrorCallBack 2719
 SetErrStatus 2717
 SetMode 2714
 VML hyperbolic functions 2658
 VML mathematical functions
 arithmetic 2583
 exponential and logarithmic 2625
 hyperbolic 2658
 power and root 2602
 rounding 2694
 special 2676
 special value notations 2582
 trigonometric 2638
 VML Mathematical Functions 2577
 VML Pack Functions 2578
 VML Pack/Unpack Functions 2707
 VML power and root functions 2602
 VML rounding functions 2694
 VML Service Functions 2714
 VML special functions 2676
 VML trigonometric functions 2638

vmlClearErrorCallBack 2722
 vmlClearErrStatus 2719
 vmlGetErrorCallBack 2722
 vmlGetErrStatus 2718
 vmlGetMode 2716
 vmlSetErrorCallBack 2719
 vmlSetErrorStatus 2717
 vmlSetMode 2714
 vmsAdd 2583
 vmsSin 2640
 vmsSub 2585
 vmzAdd 2583
 vmzSin 2640
 vmzSub 2585
 vsAdd 2583
 VSL Fortran header 2725
 VSL routines
 advanced service routines
 GetBrngProperties 2823
 RegisterBrng 2822
 convolution/correlation
 CopyTask 2867
 DeleteTask 2866
 Exec 2851
 Exec1D 2855
 ExecX 2858
 ExecX1D 2862
 NewTask 2833
 NewTask1D 2835
 NewTaskX 2837
 NewTaskX1D 2841
 SetInternalPrecision 2847
 generator routines
 Bernoulli 2808
 Beta 2798
 Binomial 2811
 Cauchy 2785
 Exponential 2777
 Gamma 2796
 Gaussian 2770
 GaussianMV 2772
 Geometric 2809
 Gumbel 2793
 Hypergeometric 2813
 Laplace 2779
 Lognormal 2790
 NegBinomial 2819
 Poisson 2815
 PoissonV 2817
 Rayleigh 2787
 Uniform (continuous) 2767
 Uniform (discrete) 2801
 UniformBits 2803
 UniformBits32 2805
 UniformBits64 2806
 Weibull 2782
 service routines
 CopyStream 2749
 CopyStreamState 2750
 DeleteStream 2749
 dNewAbstractStream 2744
 GetNumRegBrngs 2763
 GetStreamSize 2756
 GetStreamStateBrng 2762
 iNewAbstractStream 2742
 LeapfrogStream 2757
 LoadStreamF 2753
 LoadStreamM 2755
 NewStream 2739
 NewStreamEx 2741

SaveStreamF 2751
 SaveStreamM 2754
 SkipAheadStream 2760
 sNewAbstractStream 2746
 summary statistics
 Compute 2921
 DeleteTask 2922
 EditCorParameterization 2916
 EditCovCor 2897
 EditCP 2899
 EditMissingValues 2912
 EditMoments 2893
 EditOutliersDetection 2910
 EditPartialCovCor 2901
 EditPooledCovariance 2906
 EditQuantiles 2903
 EditRobustCovariance 2908
 EditStreamQuantiles 2904
 EditSums 2895
 EditTask 2884
 NewTask 2881

VSL routines: convolution/correlation
 SetInternalDecimation 2849
 SetMode 2845
 SetStart 2848
 VSL task 2725
 vslConvCopyTask 2867
 vslCorrCopyTask 2867
 vslDSScompute 2921
 vslDSSCompute 2921
 vslDSSeditcorparameterization 2916
 vslDSEditCorParameterization 2916
 vslDSSeditcovcor 2897
 vslDSEditCovCor 2897
 vslDSSeditcp 2899
 vslDSEditCP 2899
 vslDSSeditmissingvalues 2912
 vslDSEditMissingValues 2912
 vslDSSeditmoments 2893
 vslDSEditMoments 2893
 vslDSSeditoutliersdetection 2910
 vslDSEditOutliersDetection 2910
 vslDSSeditpartialcovcor 2901
 vslDSEditPartialCovCor 2901
 vslDSSeditpooledcovariance 2906
 vslDSEditPooledCovariance 2906
 vslDSSeditquantiles 2903
 vslDSEditQuantiles 2903
 vslDSSeditrobustcovariance 2908
 vslDSEditRobustCovariance 2908
 vslDSSeditstreamquantiles 2904
 vslDSEditStreamQuantiles 2904
 vslDSSeditsums 2895
 vslDSEditSums 2895
 vslDSSedittask 2884
 vslDSEditTask 2884
 vslDSSnewtask 2881
 vslDSSNewTask 2881
 vslgamma 2690
 vsLGamma 2690
 vslisssedittask 2884
 vslisSEditTask 2884
 vslLoadStreamF 2753
 vslSaveStreamF 2751
 vslssdeletetask 2922
 vslSSDeleteTask 2922
 vslsscompute 2921
 vslsSSCompute 2921
 vslsseditcorparameterization 2916
 vslsSEditCorParameterization 2916

vsLsseditcovcor 2897
 vsLsSEditCovCor 2897
 vsLsseditcp 2899
 vsLsSEditCP 2899
 vsLsseditmissingvalues 2912
 vsLsSEditMissingValues 2912
 vsLsseditmoments 2893
 vsLsSEditMoments 2893
 vsLsseditoutliersdetection 2910
 vsLsSEditOutliersDetection 2910
 vsLsseditpartialcovcor 2901
 vsLsSEditPartialCovCor 2901
 vsLsseditpooledcovariance 2906
 vsLsSEditPooledCovariance 2906
 vsLsseditquantiles 2903
 vsLsSEditQuantiles 2903
 vsLsseditrobustcovariance 2908
 vsLsSEditRobustCovariance 2908
 vsLsseditstreamquantiles 2904
 vsLsSEditStreamQuantiles 2904
 vsLsseditsums 2895
 vsLsSEditSums 2895
 vsLssedittask 2884
 vsLsSEditTask 2884
 vsLssnewtask 2881
 vsLsSSNewTask 2881
 vsPackI 2708
 vsPackM 2708
 vsPackV 2708
 vsSin 2640
 vsSub 2585
 vtgamma 2692
 vsTGamma 2692
 vsUnpackI 2711
 vsUnpackM 2711
 vsUnpackV 2711
 vzAdd 2583
 vzPackI 2708
 vzPackM 2708
 vzPackV 2708
 vzSin 2640
 vzSub 2585
 vzUnpackI 2711
 vzUnpackM 2711
 vzUnpackV 2711

W

Weibull 2782
 Wilkinson transform 2305

X

xerbla 3198
 xerbla_array 1863
 xerbla, error reporting routine 51, 2580, 3003

Z

zbbcsd 1091
 zdla_gercond_c 1792
 zdla_gercond_x 1793
 zgbcon 504
 zgbrfsx 550
 zgbsvx 686
 zgbtrs 461
 zgecon 502
 zgeqpf 806
 zgtrfs 557

zhgeqs2 1715
 zheswapr 1712
 zhetd2 1717
 zhetri2 628
 zhetri2x 632
 zhtrs2 488
 zhgeqz 1050
 zhseqr 1011
 zla_gbamv 1773
 zla_gbrcond_c 1777
 zla_gbrcond_x 1779
 zla_gbrfsx_extended 1781
 zla_gbrpvgrw 1787
 zla_geamv 1788
 zla_gerfsx_extended 1794
 zla_gerpvgrw 1829
 zla_heamv 1800
 zla_hercond_c 1802
 zla_hercond_x 1804
 zla_herfsx_extended 1805
 zla_herpvgrw 1811
 zla_lin_berr 1812
 zla_porcond_c 1815
 zla_porcond_x 1816
 zla_porfsx_extended 1818
 zla_porpvgrw 1824
 zla_syamv 1832
 zla_syrcond_c 1836
 zla_syrcond_x 1837
 zla_syrfsx_extended 1839
 zla_syrpvgrw 1845
 zla_wwaddw 1847
 zlag2c 1735
 zlapmr 1504
 zlapmt 1505
 zlarfb 1547
 zlarft 1553
 zlarscl2 1830
 zlascl2 1831
 zlat2c 1771
 zlatps 1659
 zlatrd 1661
 zlatrs 1663
 zlatrz 1667
 zlauu2 1669
 zlauum 1670
 zpbtf2 1706
 zporfsx 563
 zpotf2 1707
 zpprfs 570
 zpptrs 472
 zptts2 1708
 zrscl 1710
 zsyconv 1379
 zsyswapr 1711
 zsyswapr1 1714
 zsytf2 1719
 zsytf2_rook 1721
 zsytri2 626
 zsytri2x 630
 zsytrs2 486
 ztgex2 1725
 ztgsy2 1727
 ztrexc 1031
 ztrti2 1731
 zunbdb 1098
 zunbdb1 1671
 zunbdb2 1674
 zunbdb3 1678
 zunbdb4 1681

zunbdb5 1684
zunbdb6 1687
zuncsd 1255
zuncsd2by1 1261
zung2l 1689
zung2r 1691
zungbr 892

zungl2 1692
zungr2 1694
zunm2l 1695
zunm2r 1697
zunml2 1699
zunmr2 1701
zunmr3 1703